Landslide: Systematic Dynamic Race Detection in Kernel Space

Ben Blum

CMU-CS-12-118

May 2012

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Garth Gibson, Chair
David A. Eckhardt

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

This research was sponsored by the U.S. Army Research Office under grant number W911NF0910273.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
MAY 2012		**00-00-2012 to 00-00-2012**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Landslide: Systematic Dynamic Race Detection in Kernel Space	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Systematic exploration is an approach to finding race conditions by deterministically executing every possible interleaving of thread transitions and identifying which ones expose bugs. Current systematic exploration techniques are suitable for testing user-space programs, but are inadequate for testing kernels, where the testing framework?s control over concurrency is more complicated. We present Landslide, a systematic exploration tool for finding races in kernels. Landslide targets Pebbles, the kernel specification that students implement in the undergraduate Operating Systems course at Carnegie Mellon University (15- 410). We discuss the techniques Landslide uses to address the general challenges of kernel-level concurrency, and we evaluate its effectiveness and usability as a debugging aid. We show that our techniques make systematic testing in kernel-space feasible and that Landslide is a useful tool for doing so in the context of 15-410.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**88**	

Keywords: concurrency, kernel debugging, race conditions, runtime verification

There is a fractal nature to all aspects of life.
At the time of writing, I am focused entirely on this one project. There is a vast amount of thought in this thesis, and many more rich worlds lie hidden in each of the open avenues for future work.
Yet it pales in comparison to the scope of a Ph.D. thesis, and even more so next to all the details of a person's entire life. After I finish this project, it will become but a fond memory, a mere building block in a city of even grander things.
Human nature is beautiful in its ability to exist at any level of perspective, to hold in awareness a story of any size, and to zoom in and out at will to see how multiple parts fit together. Tearing ourselves up over minutiae is part of the fun, and contemplating overarching directions beyond our ability to change is also part of the fun.
Our true home, though, lies somewhere in between: a place of potential, and of peace.
May we always remember to return to it.

Abstract

Systematic exploration is an approach to finding race conditions by deterministically executing every possible interleaving of thread transitions and identifying which ones expose bugs. Current systematic exploration techniques are suitable for testing user-space programs, but are inadequate for testing kernels, where the testing framework's control over concurrency is more complicated.

We present Landslide, a systematic exploration tool for finding races in kernels. Landslide targets Pebbles, the kernel specification that students implement in the undergraduate Operating Systems course at Carnegie Mellon University (15-410). We discuss the techniques Landslide uses to address the general challenges of kernel-level concurrency, and we evaluate its effectiveness and usability as a debugging aid. We show that our techniques make systematic testing in kernel-space feasible and that Landslide is a useful tool for doing so in the context of 15-410.

Contents

Acknowledgements

This has been an immensely fun and fulfilling project, and it wouldn't have been possible without the contributions, guidance, and support of many people.

Big Thanks

First of all, I owe the lion's share of my gratitude equally to Garth Gibson, Jiri Simsa, and David Eckhardt.

Garth's advisorship over the past year has been excellent, from showing me how to keep my research both useful and personally satisfying, to teaching me how to share ideas with fellow researchers, to helping me think through far-off future work directions. Not working directly with his main research groups was sometimes unsettling, but validation was not in short supply: for example, it was definitely a high point when Garth emailed me unprompted asking for my input on the future of my research field.

Jiri's mentorship was invaluable, both in explanations of complicated algorithms and in general advice on good researcher habits. I would have had a much harder time understanding Dynamic Parial Order Reduction without him, and I suspect my posters and talks would have been much less put-together as well.

Working with David to share my project with the students of 15-410 really made the thesis come together. I knew I would have a rough time making Landslide available without compromising the class's integrity policies, and David helped me make it work. His advice was also crucial when designing the lecture I gave to the class to explain Landslide and recruit students for the user study; I learned more about lecture design during those two weeks than I ever thought there was to know.

Landslide's Users

I'm also highly grateful for all the patient work put in by the kernel programmers who used Landslide.

Nathaniel Filardo, as the first person to use Landslide besides myself, ran into several compatibility issues (and outright bugs) in Landslide when instrumenting his kernel. For all Landslide gave him a hard time, though, he stayed with me, was patient while I debugged, and even debugged some on his own when I was too busy myself. It was when he got Landslide

working with his kernel that I finally found confidence that it'd be able to work on arbitrary student kernels.

Thanks also to the other members (and former members) of 15-410 staff who used Landslide on their kernels to help me prepare for working with the students: Josiah Boning, Alex Crichton, and Michael Sullivan.

A special form of thanks to the students I worked with, who not only had to navigate Landslide's interfaces but also had to learn the underlying concepts as well, all while under time pressure to finish their kernel projects at the same time. To Nadim Taha, Margaret Schervish, Timothy Passaro, Joon-Sup Han, Pranjal Jumde, and others who did not ask to be mentioned by name: I could not have evaluated Landslide without you.

Another piece of thanks goes to Nadim Taha for his above-and-beyond investigation of the decision trees Landslide explored using his kernel, which resulted in the insights presented in Section 8.7.2.

Collaboration

Thanks to Wind River for Simics™, the simulation software for which Landslide is built. While developing code to fit into larger infrastructure always has its hairy points, Simics's interfaces were generally pleasant, and the expressive power of operating in its simulation environment was invaluable.

Thanks to the members and companies of the PDL Consortium (including Actifo, APC, EMC, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard, Hitachi, Huawei Technologies, Intel, Microsoft, NEC, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMware, and Western Digital) for their interest, insights, feedback, and support. It has been a pleasure to work in this research group.

Support

More gratitude goes to the people who helped me revise my thesis document and defence talk. While Garth and David provided the most detailed criticism, many of my friends helped me polish my work: Wolfgang Richter, Carlo Angiuli, Joshua Wise, Ryan Pearl, and Michael Sullivan critiqued this document, and Jiri Simsa, Matthew Maurer, Greg Hanneman, and Carlo Angiuli critiqued my defence presentation.

Finally, huge thanks to Deborah Cavlovich and Catherine Copetas, whose organisational efforts carried me through the Master's program. These are the people who make the gears turn under the hood.

I hope you enjoy reading some or all of this.

Chapter 1

Introduction

Race conditions are notoriously difficult to debug. Because of their nondeterministic nature, they frequently do not manifest at all during testing, and when they do manifest, it is difficult to reproduce them reliably enough to collect enough information to help debugging.

Many techniques exist for dynamic testing of concurrent systems for race conditions. Systematic exploration [God97], the strategy we focus on in this work, involves making educated guesses as to what points during execution a preemption would be most likely to expose a bug, enumerating the different possibilities for interleaving threads around these points, and forcing the system to execute all such interleavings to check if any of them results in incorrect behaviour. Systematic exploration provides a better alternative to conventional long-running stress tests, because it is less likely to overlook buggy execution patterns, and it enables a testing framework to report more thorough debugging information. Compared to other dynamic analyses, such as data race detection [EMBO10], systematic exploration is able to find a wider range of types of concurrency errors because of its ability to manipulate the execution of the system under test.

In kernel space, race condition debugging becomes even more difficult. Many aspects of the concurrency implementation itself are part of the system being tested. While user-space testing frameworks rely on the underlying kernel to provide common concurrency abstractions which the framework can use to drive the test, systematic exploration in the kernel itself necessarily exists underneath such abstractions. Furthermore, a tool that supports testing multiple kernels must support many different possible scheduler designs.

The claim of this dissertation is as follows.

> **Systematic exploration in kernel space is possible by understanding a kernel's concurrency through instrumentation of its internal abstractions, and can be made efficient by relying on the user to control the scope of the test.**

We present Landslide,[1] a tool for applying systematic race detection techniques to kernel-level code. It is geared towards kernels that implement the Pebbles specification, the main

[1] Landslide *(n)*: a phenomenon which demonstrates that Pebbles are not as stable as you might think.

project in Operating System Design and Implementation (15-410) at CMU, and is implemented as a module for Wind River Simics™ [MCE+02], the x86 simulator that students use to run their Pebbles kernels. In order to use Landslide, a kernel programmer must instrument their kernel to inform Landslide of important concurrency events during the execution, and configure Landslide to focus its search on components of the kernel most relevant to the test. Using this instrumentation and configuration, Landslide is able to enumerate all possible interleavings of kernel threads at a granularity both fine-grained enough to expose interesting concurrency behaviour and coarse-grained enough to produce state spaces that are feasible to explore. Landslide then forces the kernel to exercise each of these interleavings, and checks for race conditions in each of them. When Landslide finds a bug (predicted by a set of common checks and heuristics), it stops execution and prints information about the sequence of interleavings that exposed the bug.

In this work, we study the challenges inherent in applying systematic testing techniques in kernel-space, in contrast with user-space applications (Chapter 3), define common terms used in the rest of this document (Chapter 4), present our techniques (some sound, and some heuristic) for addressing them (Chapter 5), and document the process of using Landslide (Chapter 6). We evaluate Landslide both by studying its potential to be a helpful debugging aid in 15-410 and by studying certain bugs in detail to determine effective usage strategies (Chapter 7). Finally, we discuss possible avenues for future work in education, use of additional testing techniques, application to general purpose kernels, long-running testing approaches, and theoretical study of systematic exploration (Chapter 8).

Chapter 2

Related Work

Compared to conventional stress testing, systematic exploration is a relatively young approach to concurrency verification [God97]. Recent work has focused on theoretical aspects of the technique, on distributed systems, and in userspace in general. Other work has focused on kernel verification apart from systematic exploration, using techniques such as data race detection and formal model verification. Still other efforts are for testing and verification techniques that are orthogonal to systematic exploration and kernel-level testing entirely. Here we discuss related work in all three categories.

Compared to related systematic testing tools in its field, Landslide is more of an exploratory work, presenting techniques to make systematic testing compatible with kernel-level code, rather than a development on the core testing approach itself.

2.1 Systematic Exploration

Dynamic Partial Order Reduction (DPOR) [FG05, YCGK07, YCGK08, SBGH11] is an algorithm for reducing the state space of "all possible thread interleavings" by identifying independent thread transitions and pruning the redundant state spaces that result from interleaving those transitions. Landslide makes generous use of the DPOR algorithm.

dBug [SBG10] is a systematic testing tool which makes use of DPOR. It focuses on multithreaded userspace applications, identifying when to preempt by interposing on `libc` and `pthreads` library calls. It uses a message-passing model for inter-thread communication and for establishing a happens-before relationship between transitions. The dBug project inspired the development of Landslide.

CHESS [MQB+08] is another userspace systematic testing tool. The authors explore the insight that many races require very few forced preemptions to uncover, and develop a search strategy which prioritises thread interleavings with fewer preemptions. Landslide does not (yet) make use of this strategy, though we did find a related insight which we discuss in Section 8.7.1.

There are also several tools, such as MaceMC [KAJV07] and MODIST [YCW+09], which provide systematic testing frameworks for networked and distributed applications. MODIST is especially notable in its ability to test unmodified distributed systems by using event interposition.

RacePRO [LVT+11] is a tool based in kernel space which uncovers "process races" (races involving interleaving system call patterns between multiple processes) using systematic exploration of multi-process interactions in user-space. They use system calls, such as filesystem accesses, as points to explore different interleavings of concurrent code, and detect bugs which can corrupt persistent system resources.

Finally, DeMeter [GWZ+11] is a more recent tool for systematic exploration that introduces Dynamic Interface Reduction as a strategy for constraining the size of the state space, which means targetting the testing technique to only one component of the program at a time. Landslide makes use of the same general concept in its user interface (Chapter 6) by providing the user with options to constrain decision point identification to certain subsets of the kernel.

2.2 Kernel-level Verification

DataCollider [EMBO10] is a tool for detecting data races, a special kind of concurrency bug which involves concurrent unprotected accesses to memory that can interleave unsafely on a fine-grained level. DataCollider uses memory access sampling, and remains largely agnostic of synchronisation operations, to find data races in the Windows 7 kernel. In Section 8.3 we discuss how systematic execution might be enhanced by integrating data race detection techniques.

Carburizer [KRS09] is an effort to make imperfect kernel-level drivers work more reliably in the presence of faulty peripheral devices. They use static analysis to identify variables "tainted" by device input and unsafe uses thereof. Carburizer is able to transform such badly-written code to be more robust. Kernel device drivers are a ripe breeding ground for bugs, because they are frequently written by less careful developers, and the sheer volume of code makes them difficult to rigorously review. Due to the interrupt-driven nature of many drivers, they are also prone to concurrency bugs, which Carburizer does not address.

SimTester [YSaR12] is another tool for testing kernel device drivers. Like Landslide, SimTester uses Simics as its execution environment, and identifies key decision points during execution at which to inject interrupts. SimTester focuses on races involving device interrupt handling code, whereas Landslide is geared towards non-driver inter-thread races. Additionally, SimTester's testing model involves forcing only one interrupt per test run, which is a lighter-weight approach than systematic exploration.

In Section 8.4.4 we discuss the potential applicability of systematic exploration to concurrency bugs in kernel device drivers.

2.2.1 New Kernel Architectures

On the far end of the "formality and correctness" spectrum for kernel verification lies seL4 [KEH+09], a microkernel fully designed and specified in Haskell and translated directly into C. Because Haskell is type-safe (strict constraints on legal program behaviour that make it easier to reason about) and pure (most code cannot have stateful side-effects), the developers of seL4 were able to formally prove the specification's correctness and the implementation's adherence to the spec. Developing kernel code in languages and environments that enable formal complete verification, even in small components of a larger codebase (generally more tractable than seL4's extreme approach), represents an admirable step towards writing correct code to begin with and avoiding many classes of bugs entirely.

Barrelfish [BBD+09] is another novel kernel architecture designed with concurrency in mind. The Barrelfish kernel relies exclusively on message-passing for inter-thread communication, for increased performance on massively multi-core systems. While not a verification project, the strict limitation of inter-thread communication makes the concurrent properties of the system easy to reason about in ways that modern production kernels do not provide. We believe Barrelfish, like seL4, is a step in the right direction for bringing better concurrent code into the world.

However, in both cases, such steps have little bearing on code that exists in the world today, and it is also important to find ways to verify more imprecisely-written code that already has bugs, which is the focus of our work.

2.3 Orthogonal Testing Techniques

Research in dynamic verification has also seen other techniques apart from systematic exploration.

Data race detection is the canonical runtime concurrency verification technique. Tools such as ThreadSanitizer [SI09] track synchronisation operations and happens-before relationships to identify unsafe concurrent memory accesses. Such accesses, called "data races", might not necessarily cause incorrect behaviour, but are usually warning indicators of such, and represent bad coding practice in any case. RaceTrack [YRC05] is another data race detection tool notable for its ability to adaptively adjust its testing granularity after some execution analysis.

Symbolic execution [Kin76, YST+06] is a technique for testing programs by abstractly interpreting symbols and operations within a program, rather than directly executing its compiled code, which grants the ability to analyse conditional statements and cause-effect relationships among certain code paths. Tools such as KLEE [CDE08] and projects such as Automated Exploit Generation [ACHB11] demonstrate the effectiveness of symbolic execution in dynamic verification.

There has also been recent research in searching for which conditions, among a known

set of conditions associated with a buggy behaviour, are actually necessary to cause bugs to arise [SKM+11]. Such strategies are helpful because even once a bug is exposed, it can be difficult to determine what specifically went wrong.

We discuss the possibility of integrating each of these orthogonal techniques into a testing framework such as Landslide in Section 8.3.

Deterministic multithreading is a different technique for dealing with concurrency bugs which focuses on avoiding them when running production code rather than trying to expose them during testing. Deterministic multithreading tools, such as Kendo [OAA09], Peregrine [CWG+11], and DThreads [LCB11], analyse the execution of a concurrent system, compute particular scheduling patterns which will not produce buggy behaviours, and force the system to follow those schedules while maintaining good parallel performance. Deterministic multithreading serves a different purpose than systematic exploration: it aims to ensure that code already running in the real world does not encounter concurrency bugs even though they might exist, while our goals are instead to uncover such bugs and help fix them beforehand.

Chapter 3

Challenges of Kernel Space

In this chapter, we discuss several problems that challenge systematic exploration's applicability to kernel space. We address these problems partly with certain aspects of Landslide's design and partly by relying on the user to provide important information, which we discuss in Chapter 5 and Chapter 6 respectively.

3.1 Causes of Concurrency

In user space, a systematic exploration tool may cooperate with the underlying kernel to help control the concurrent behaviour of the system [SBG10]. Simple system call invocations can cause a particular thread to run at a particular time, or to block while another thread runs first.

In kernel space, however, the scheduler is part of the system being tested, and we can no longer always interrupt the execution of a test case to ask the scheduler to instantly start running a different thread.

- **Non-preemptibility.** Certain regions of code may be non-preemptible, so a testing tool must know when it is legal to preempt a kernel thread.
- **The context switcher.** A context switch between threads is no longer "instantaneous": many instructions must be run between when we decide to preempt and when the next thread begins running, and the tool must be aware of this "intermediate state". A tool must be aware of the context switch process both to avoid flooding the kernel with interrupt frames that would overflow the stack, and to be able to ignore shared memory conflicts inherent to all thread transitions (Section 5.4.2).
- **Run-queue tracking.** The kernel's scheduler and the tool must cooperate in some way so that the tool can both know what threads are runnable at every point during execution and cause any given runnable thread to begin running in place of the current one.

Figure 3.1: *¡Cuidado! ¡Las llamas son muy peligrosas!* [Eck11]

3.2 Ad-hoc Thread Communication

Inter-thread communication can be much more ad-hoc in kernel-space than in user-space [EMBO10], and a systematic testing framework needs to be aware of all the kernel's synchronisation idioms.

- **Avoiding reliance on message-passing.** Some systems for user-space systematic testing require that threads communicate only by message-passing, to better track the concurrency relationships between thread transitions [SBG10]. With few exceptions [BBD+09], kernels do not rely on message-passing as a primary communication mechanism, and to be compatible with the kernels of today, a testing framework must allow for less idealised state-sharing (Section 5.2.3).

- **Recognising blocked threads.** It is difficult to detect when threads are ready to do useful work. Even if notionally runnable, they may be waiting in a yield loop (Sections 6.2.1 and 8.4), which can be difficult to discern without an understanding of whatever shared resource the thread is waiting for. A tool that did not recognise this might decide to attempt to run that thread over and over, and get stuck because the other threads would never be allowed to make progress.

- **Thread lifecycle tracking.** It is also difficult to demarcate threads' lifecycles: when a new thread gets created, when is it available to run? When a thread is exiting, at what

point does it stop running code (Section 6.2.1)? In user-space, these boundaries are defined by the system call instructions, but in kernel-space, the boundary is fuzzier.

- **Use of virtual memory.** The kernel's complete control over the machine's virtual memory system means a system for tracking memory accesses must be aware of changing virtual memory mappings (Section 6.1). A tool not aware of virtual memory might falsely identify whether or not two transitions' memory accesses conflict.

3.3 Kernel Design Independence

One challenge particular to the context of 15-410 is that the kernels Landslide must be able to test may all use slightly different implementations to achieve the same goals. While 15-410's kernel project mandates that students implement the Pebbles specification, students are free to make their own design decisions under the hood, which can result in significantly different scheduler behaviour among multiple kernels. In general, Landslide may rely on concrete details provided by kernel specifications, but must be compatible with arbitrary implementation details. For Landslide to be generally applicable to many different kernels, even only multiple ones that implement the same specification (such as Pebbles), it must make some abstract assumptions about the kernel design which are compatible with many different implementations.

We built Landslide to be compatible with multiple designs for the following major scheduling behaviours:

- **Runqueues**: Does the kernel store the currently-running thread on the runqueue, or is it "checked out" while running and stored separately?

- **Mutexes**: When a thread blocks on a mutex, is it left on the runqueue in a yield loop, or is it explicitly descheduled (e.g., moved from the runqueue to another queue)? In yield-looping mutexes, when do blocked threads become "unblocked", notionally? (This might happen before the blocked thread runs next.)

- **Idling**: Does the kernel have an explicit idle thread, or is there an idle loop that runs on the stack of whatever thread was last running? Do explicit idle threads run their idle loop in userspace or in kernelspace?

- **Thread Creation**: When a thread is newly forked, is it placed onto the runqueue for later, or is it context-switched to immediately? Do just-forked threads begin life through the usual context-switch-return path, or is there a special path for that?

- **Test Lifecycle**: Apart from running on a particular input that the user wishes to test, the kernel may need to perform extra work, such as initialisation (e.g., the bootup process) and housekeeping (e.g., cleaning up dead processes). Though the entire kernel

is notionally the system under test, the user-space test case expresses what the user is actually interested in, and some of this extra work is irrelevant. Which parts of this work should be included in or excluded from the test?

Chapter 4

Terminology

In this chapter we define certain terms that have specific meanings in the context of Landslide.

4.1 Basic Terms

1. **Guest kernel**: The kernel which is being tested for concurrency bugs by Landslide.
2. **Test case**: Abstractly, the set of inputs under which the guest kernel is tested. Practically, a user-space program which runs on top of the guest kernel to execute a particular set of system calls, while Landslide performs systematic exploration. The expected results of the test case's system call invocation pattern informally define the kernel specification.

4.2 Scheduling Terms

The following terms refer to parts of the guest kernel's execution.

3. **Thread**: One participating agent in a concurrent program, which executes code sequentially, with the potential to be interleaved with the execution of other threads in the system. Each thread has a unique numeric identifier (TID).
4. **Involuntary preemption**: A context switch from one thread to another caused by a nondeterministic event, such as a timer tick or device interrupt. The systematic testing framework must control all sources of nondeterminism. In this work, we focus only on timer-driven thread switches.
5. **Voluntary reschedule**: A context switch from one thread to another which the guest kernel performed automatically, not triggered by nondeterministic events and/or Landslide but rather as part of its normal execution.[1] See Section 5.2.4.

[1]Voluntary reschedules may or may not be necessary for the kernel to behave correctly at all (for example: switching away from an exiting thread or from a thread blocked on `wait` is necessary, but switching to a newly-

4.3 Systematic Exploration Terms

The following terms refer to parts of Landslide's analysis, regardless of the implementation of the kernel under test.

6. **Decision point**: A point during execution (between two consecutive instructions, precisely) which is deemed "interesting" in terms of the likelihood that thread switches at that point will cause concurrency bugs to arise. Every thread switch occurs at a decision point; some are voluntary reschedules caused by the kernel and others are involuntary preemptions caused by Landslide.

7. **Transition**: A sequence of instructions executed by a single thread between two decision points. At a given decision point, Landslide chooses which thread will run (and causes it to do so), and that thread executes a transition, after which point Landslide will have identified a subsequent decision point

8. **Decision set** (or **Set of decision points**): One or more predicates on the execution state of the guest kernel which identify whether any current state should be a decision point. (For example, when we say "the decision set including all calls to `mutex_lock`", the corresponding predicate is simply "did the kernel just invoke `mutex_lock`?".)

9. **Decision tree** (or **Execution tree**): The tree defined by the possibility at each decision point of causing any runnable thread to execute its own transition. Abstractly, this tree comprises all possible states reachable by any interleaving of threads. A decision tree is created/discovered during an exploration using a particular decision point set; the set is what the user configures, and the tree is what arises as a result.

   Figure 4.1 depicts a decision tree for a simple race with three possible interleavings.

10. **Branch**: One particular execution of the test case; a set of decision points with exactly one transition between each of them, characterising a single interleaving of threads.

11. **Interleaving**: A less precise / more abstract term for a branch, or for a subset of transitions which make up a branch.

12. **Bug**: An execution state which Landslide identifies as incorrect, which is indicative of illegal behaviour. Each branch of the tree may or may not have a bug in it. See Section 5.3.

13. **Backtracking**: Performed at the end of each branch of the tree, when the test case has finished running. Landslide identifies a decision point from the current branch at which the next interleaving to explore diverges from the current one, reverts the state of the guest kernel to what it was at that decision point, and causes a different thread to run, hence exploring a different branch/interleaving.

forked thread is not). Involuntary preemptions, however, are never necessary for correct kernel behaviour.

14. **Decision trace**: A list of information about all decision points in a given branch, printed when a bug is found to help the user analyse its cause. The trace contains, for each decision point, the TID of the thread that was running previously, the TID that was newly chosen, the current instruction pointer, and the stack trace of the previous thread at the point it was switched away from.[2]

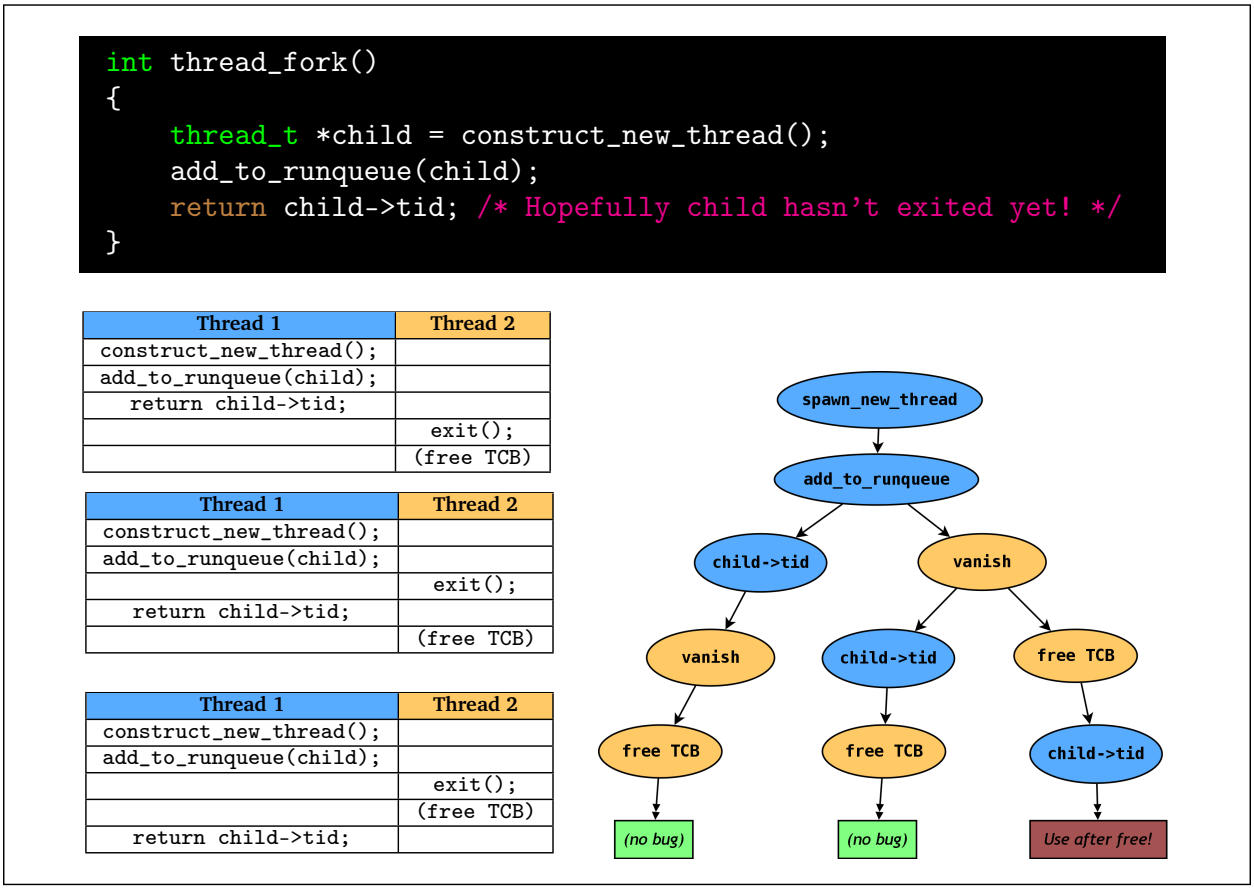


Figure 4.1: This `thread_fork` implementation is prone to a use-after-free bug. Three different possible interleavings with the newly-forked thread (left) can be viewed together as a single decision tree (right). (Section 7.2.1 studies this particular bug in detail.)

[2]Though Landslide does not provide it, a decision trace could also contain for each transition a list of shared memory accesses that conflicted with other transitions, and/or a listing of happens-before relationships between transitions.

Chapter 5

Design and Implementation

In this chapter we describe Landslide's implementation in detail. Section 5.1 outlines the testing model on top of which Landslide's exploration mechanisms are built. Section 5.2 describes the individual components within Landslide. Section 5.3 describes Landslide's metrics for identifying bugs. Section 5.4 describes how Landslide achieves effective state space reduction. Section 5.5 describes the debugging feedback Landslide provides to the user when bugs are encountered.

5.1 Landslide's View of the World

Landslide's model for applying systematic exploration in kernel space is comprised of four key points, which we overview here.

5.1.1 Simulated Execution

Landslide is implemented as a module for Simics [MCE+02], a full-system x86 emulator. When running the kernel, Simics calls into Landslide once every time the kernel executes an instruction or performs a memory read or write. Landslide uses this information, in conjunction with the user-provided instrumentation, to maintain its internal representations of the state of the guest kernel.

Landslide makes use of Simics "bookmarks", a feature which enables checkpointing and restoring the execution state of the guest kernel, to implement backtracking when the end of each branch in the decision tree is reached.

5.1.2 Timer-Driven Scheduling

In this work, we focus specifically on nondeterministic scheduling driven by timer interrupts. Landslide assumes that timer interrupts are the only source of nondeterminism for the guest

kernel, so controlling when they occur theoretically allows for complete control over the concurrent behaviour of the test case. In future work (Section 8.4.4), we also address causes of non-determinism more complex than timer-driven thread scheduling, such as interrupts and data I/O from peripheral devices.

Landslide needs to impose some requirements on the guest kernel's scheduling behaviour, in order for its control over scheduling to work:

- **Timer ticks control "runnable" threads.** With the exception of a non-preemptible "scheduler lock", and yield-looping mutexes (both of which must be instrumented by the user), a thread's presence on the runqueue indicates that a finite number of timer interrupts in succession will eventually cause it to run.

  Landslide treats the sleep queue no differently from the runqueue when deciding which threads are runnable. The guest kernel will treat it differently, but since sleeping for a predetermined amount of time is never an appropriate way to solve race conditions, Landslide treats sleeping threads as notionally runnable. This fits directly into its model that runnable threads are ones that can be caused to run with a finite number of timer ticks in succession.

- **No idling when progress can be made.** The kernel must not enter its idle loop (whether in an explicit idle thread or not) when the kernel is not truly idle. Landslide uses the idle loop to detect when a test begins/ends (test lifecycle tracking) and to detect when all threads in a test are wedged (bug detection).

Section 5.2.2 gives more detail about how Landslide abstracts raw timer interrupts into the ability to cause arbitrary threads to run.

5.1.3 False-Negative-Oriented Bug Detection

Without a formal specification of the internals of the guest kernel's implementation [KEH+09], it is impossible to identify both soundly and completely when a behaviour that constitutes a "bug" arises during a test case's execution. Landslide's bug reporting is false-negative oriented, meaning that it does not check for suspicious behaviours that might indicate underlying bugs, so it may report that it found no bugs even if some existed. If Landslide does report a bug, though, it is almost certainly correct. Section 5.3 details the conditions Landslide uses to identify when something has gone wrong.

5.1.4 User-Assisted State Space Reduction

In addition to Dynamic Partial Order Reduction (DPOR) for automated state space reduction, Landslide relies on the user's guidance to help mitigate the possible combinatorial explosion of thread interleaving possibilities. This manifests in two ways:

- **Iterative refinement of interleaving granularity.** The recommended usage pattern is to begin by searching state spaces with coarse-grained interleavings, and then refining the decision set afterwards until bugs are found. Landslide automatically identifies a "minimal decision set", comprising only voluntary reschedules (Section 5.2.4), to use as the coarsest granularity.

- **Decision point selection.** Landslide provides an interface for the user to manually limit identification of decision points to only "relevant" modules of the kernel, thereby generating thread interleavings only around points within those modules.

- **Memory conflict selection.** Landslide also provides a mechanism for ignoring shared memory conflicts on certain common data structures (especially those accessed in every thread transition, such as the scheduler queues), to enable DPOR to achieve greater reduction. Intuitively, this represents sacrificing the ability to find races around those data structures to make searching other components more feasible.

We advocate relying on the user for such information because the user need only understand the basics of their kernel design (which is difficult for Landslide to guess) while not knowing in advance which bugs are being looked for (which Landslide is able to provide). We claim that this combination of the user's knowledge and Landslide's testing mechanisms leads to an effective usage dynamic, in which the user "steers" Landslide towards focused search spaces that are more likely to find bugs.

The interfaces by which the user provides this information are documented in Section 6.3.

5.2 Components of Landslide

In this section we briefly describe each major component of Landslide.

5.2.1 Kernel Instrumentation

The kernel instrumentation serves as the glue between what the guest kernel is doing and Landslide's understanding of the guest kernel's state.

- **User-provided instrumentation.** Some parts of the kernel may be written in any number of ways, and hence require the user's assistance for Landslide to understand. The user-provided instrumentation, described in detail in Section 6.2, informs Landslide about thread-related lifecycle and scheduling events and the anatomy of the kernel's scheduler.

- **Automatic instrumentation.** All Pebbles kernels have some things in common (largely due to the common starter code provided for the class project), and Landslide's build

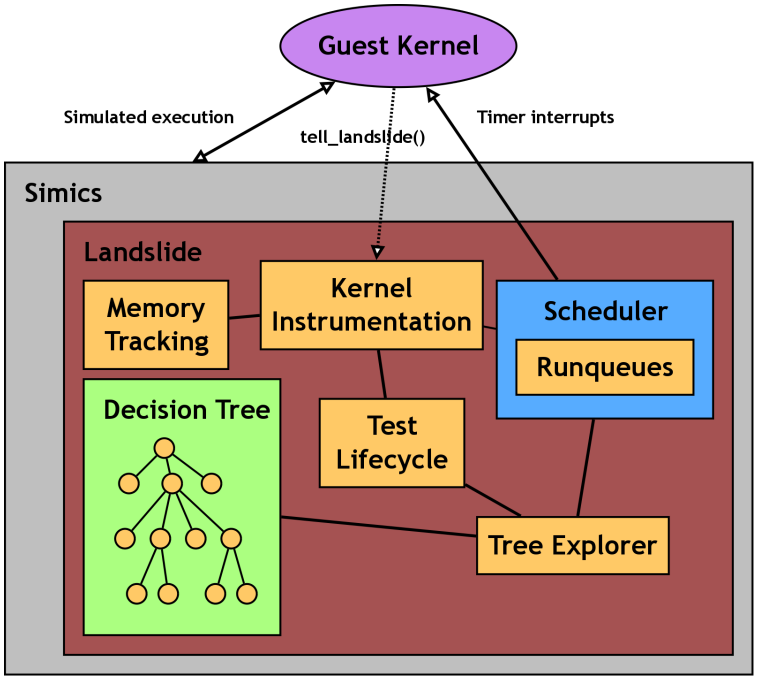


Figure 5.1: A simple visualisation of Landslide and its components, and how they interact with the guest kernel.

system is able to automatically instrument certain parts of the kernel. This instrumentation informs Landslide about the dynamic memory allocator, common library functions such as `panic`, and certain aspects of the kernel's executable format.

5.2.2 Scheduling

The Landslide scheduler is responsible for keeping track of which threads exist in the guest kernel: which are runnable at any given time, and when they are created and destroyed.

It maintains a "mirror image" of the guest kernel's scheduler state in the form of three queues, a pointer to the currently-running thread, and a pointer to the previously-running thread. The queues are the *runqueue*, containing the runnable threads, the *sleep queue*, containing threads which become runnable after a certain number of timer ticks, and the *deschedule queue*, which might not correspond to a data structure in the guest kernel, but contains all other threads that exist on the system, which are not runnable for whatever reason.

The Landslide scheduler also tracks which important actions each thread is performing. These actions are *forking*, *vanishing*, and *sleeping*, which are described in the next section,[1] and also *in_timer_handler* and *in_context_switch*, which express what type of context switch a given thread may be performing (voluntary reschedule, etc), and are useful for both thread

[1]Note that "vanish" is simply Pebbles's name for "exit".

Algorithm 1 Landslide's scheduler's routines for tracking thread lifecycles. These routines are invoked each time the guest kernel calls one of the `tell_landslide` annotations.

```
Per-thread state.
bool forking, sleeping, vanishing, in_timer_handler;
int current_tid;

function HANDLE_TELL_LANDSLIDE_FORKING
    forking ← true;
end function

function HANDLE_TELL_LANDSLIDE_SLEEPING
    sleeping ← true;
end function

function HANDLE_TELL_LANDSLIDE_VANISHING
    vanishing ← true;
end function

function HANDLE_TELL_LANDSLIDE_THREAD_SWITCH(int new_tid)
    if !in_timer_handler then
        if sleeping then
            ADD_TO_SLEEP_QUEUE(current_thread);
            sleeping ← false;
        else if vanishing then
            DESTROY_THREAD(current_tid);
            vanishing ← false;
        else if forking then
            CREATE_THREAD(new_tid);
            forking ← false;
        end if
    end if
    UPDATE_CURRENT_THREAD(new_tid);
end function
```

scheduling (Section 5.2.2) and detecting voluntary reschedules (Section 5.2.4).[2]

Thread Lifecycle Tracking

The Landslide scheduler relies on the guest kernel to inform it about certain important events in the thread lifecycle. The guest kernel does this by means of annotations (which are described in Section 6.2.1). Algorithm 1 shows how Landslide handles these annotations internally.[3]

[2]There are also other minor actions: *readlining*, used to track test lifecycle (Section 5.2.6), *just_forked*, used for special-case context switch behaviour (Section 5.2.4), and also several flags for tracking mutex lock/unlock events.

[3]There are also cases for the runqueue annotations, `tell_landslide_on_rq` and `tell_landslide_off_rq`, which we omit for brevity. One important note is that the routine for

Algorithm 2 Landslide's scheduling algorithm. This procedure for updating Landslide's state is executed once per instruction, with a corresponding value for `pc` (the program counter) each time. The predicates on `pc` are part of the kernel instrumentation (Section 5.2.1).

```
Global scheduler state.
bool schedule_in_flight;
int target_tid;

Per-thread state. (Updated elsewhere.)
bool in_timer_handler, in_context_switch;
int current_tid;

function SCHEDULER_UPDATE(int pc)
    if schedule_in_flight then
        ASSERT(in_timer_handler || in_context_switch);
        if KERNEL_EXITING_TIMER(pc) || (!handling_timer && KERNEL_EXITING_CONTEXT_SWITCH(pc)) then
            The kernel has just finished rescheduling and is about to resume normal thread execution.
            if current_tid != target_tid then
                The kernel switched to an undesirable thread. Keep the schedule operation "in-flight".
                CAUSE_TIMER_INTERRUPT();
            else
                The in-flight schedule is "landing".
                schedule_in_flight ← false;
            end if
        end if
    else if NEED_TO_PREEMPT() then
        target_tid ← CHOOSE_NEW_THREAD();
        schedule_in_flight ← true;
        CAUSE_TIMER_INTERRUPT();
    end if
end function
```

Thread Scheduling

Finally, the Landslide scheduler is responsible for managing involuntary preemptions, causing arbitrary threads to begin running in place of the current one (as chosen at decision points and the ends of branches by the explorer, Section 5.2.5).

Though we define timer interrupts as the only source of non-determinism in our environment, it is more useful to view the concurrent behaviour with a higher-level abstraction, in terms of the set of runnable threads and the ability to preempt the currently-running thread with any different runnable one.

Landslide's scheduling technique, called the *schedule in-flight*, involves successively injecting timer interrupts to trigger context switches until the desired thread begins to run. Algorithm 2 shows how Landslide makes this happen.

One alternative simpler method would be, when triggering a timer interrupt, to tell the

`tell_landslide_on_rq` must also check the "forking" flag, in case the kernel's `fork` implementation does not immediately switch to the new thread but instead adds it to the runqueue for later.

guest kernel explicitly which thread should be run next. This approach could potentially save the cost of extra context switches, especially in test cases with very many threads. However, it would also require the kernel programmer to write extra code in their timer handler and/or context switcher. Because we mostly focus on test cases with few threads, we judged that ease of use was more important than the marginal performance increase, so we chose the "in-flight" approach instead.

5.2.3 Memory Tracking

Landslide maintains a mirror image of the guest kernel's dynamic allocation heap, so it can know at any point which memory ranges are allocated and which ranges used to be allocated but now are freed. This set is updated each time the guest kernel calls `malloc` or `free`. This heap tracking provides the ability to check for dynamic allocation errors (e.g., use-after-free and double-free bugs), in a similar fashion to Valgrind [NS07].

Landslide also maintains a set of shared memory accesses made since the last decision point, for use with Partial Order Reduction (Section 5.4.2). Whenever the guest kernel accesses memory in its heap or in its global data regions, Landslide adds the address of the access to the set, with a flag indicating whether it was a read or a write. (If the address was already present, and the recorded access was a read and the current access is a write, we upgrade the recorded access to a write. Otherwise if the address was already present, we do nothing.)

Landslide ignores shared memory accesses from the kernel's dynamic allocator itself, and it also ignores shared memory accesses from the components of the kernel's scheduler which run every transition (Section 5.4.2).

When Landslide reaches a decision point, this accumulated set is copied in the decision tree (Section 5.2.5), and reset to empty before continuing execution.

5.2.4 The Arbiter

The arbiter identifies points during execution that should count as decision points. The selection is mainly controlled by the user, during the annotation and configuration process. In addition, the arbiter also automatically identifies *voluntary reschedules*, which comprise the "minimal necessary set" of decision points.

In Landslide, it is always necessary to identify a decision point during a voluntary reschedule, to maintain an invariant that each transition between decision points is comprised of the execution of only one thread. In future work, we may also investigate the implications of allowing some transitions to consist of multiple threads' executions, which would have applications for heuristic state space reduction.

Because voluntary reschedules happen of the guest kernel's own volition, when a thread begins one, it will necessarily not have been preempted by the timer handler. Hence, we iden-

tify voluntary reschedules by determining when a thread enters the context switcher without having entered the timer handler.[4]

In future work (Section 8.3), the arbiter may also automatically identify extra decision points, such as conflicting shared memory accesses.

5.2.5 The Explorer

The explorer maintains a representation of the current branch of the decision tree. It is responsible for checkpointing the state of both Landslide and the guest kernel at each decision point, deciding at the end of the test which branch of the tree to execute next (i.e., selecting which decision point should have been decided differently), and backtracking to appropriate points in the test's execution.

At each decision point, Landslide creates a new node in the decision tree. It stores the TID of the thread that was chosen, the state of the scheduler (which threads are on the runqueue), the state of the heap and the accumulated set of shared memory accesses, and a stack trace of the thread that was running.

5.2.6 Test Lifecycle

Landslide knows the state of the test case through a simple state machine, which is updated with information about the number of threads currently on the system, the number of threads on the runqueue, the state of the idle thread (if it exists), and whether or not the shell is waiting for keyboard input.

The state machine relies on certain information provided by the scheduler (Section 5.2.2).

- **Is the shell waiting for keyboard input?** In general, the shell waits for keyboard input only when the kernel is ready for the user (or Landslide) to cause a test to run, or after the test has finished and a new one could be run. However, the shell may sometimes be waiting for keyboard input while other threads have some work to do (for example, the init process might not yet have reaped a reparented child process), and this work may be affected by race conditions during the test, so we need to include it in the test as well. For these cases an additional check is needed.
- **Is the kernel idling?** This indicates whether any work is left to be done during bootup or during teardown.
- **How many threads currently exist on the system?** This indicates whether the test has begun running.

[4]Landslide actually identifies voluntary reschedules at the point when the reschedule ends, so the *previous* thread would not have entered the timer handler, and the current thread may or may not have. We do this to avoid the complexity of dealing with the possibility that the guest kernel would have interrupts disabled at the beginning of the reschedule and not re-enable them until after the thread switch.

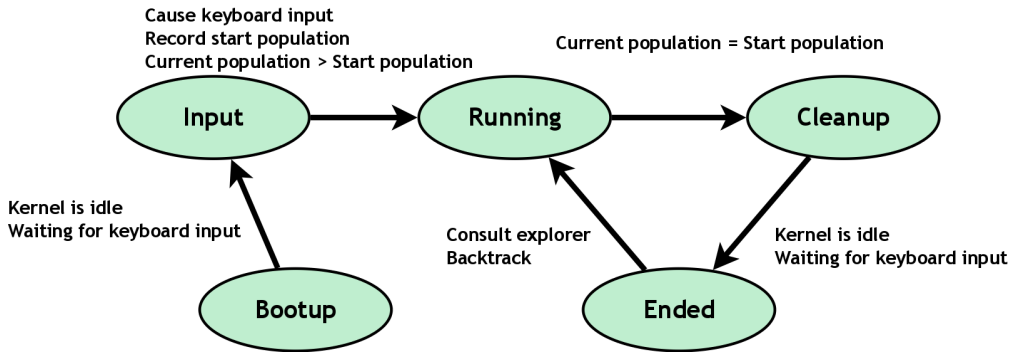


Figure 5.2: The test lifecycle state machine.

The states in the test lifecycle, and the rules for transitioning between them, are as follows. Figure 5.2 depicts the states and transitions visually.

1. **Bootup.** Initial state. When the kernel is idling and the shell is waiting for keyboard input, advance state.

2. **Keyboard input.** Record the current thread count (the "start population"). Generate keystrokes to cause the desired test case to run. When the current thread count is greater than the start population, advance state.

3. **Test running.** Begin constructing the decision tree here (since it is pointless to backtrack to before the test begins). When the current thread count equals the start population, advance state.

4. **Test cleanup.** The test processes have exited, but some housekeeping may yet remain. When the the kernel is idling and the shell is waiting for keyboard input, advance state.

5. **Test ended.** Invoke the explorer (Section 5.2.5) to decide where to rewind to, and backtrack (to step 3).

5.3 Identifying Bugs

In order to identify when the guest kernel has done something incorrect, Landslide performs several different types of checks, some accurate but noncomprehensive, and some heuristic-based.

5.3.1 Definite Bug-Detection Conditions

1. **Kernel panic bugs.** If the kernel invoked `panic`, it detected its own bug, and Landslide need do nothing but report it.

2. **Use-after-free bugs.** Whenever the kernel accesses memory in the heap (when not in the dynamic allocator itself), Landslide verifies that the address is within an allocated range. If not, Landslide proclaims the access to be illegal.[5]

3. **Deadlock bugs.** If Landslide finds no runnable threads on the runqueue,[6] or if it detects a cycle of threads blocked on each other, it declares that the kernel has deadlocked.

5.3.2 Probable Bug-Detection Conditions

4. **Memory leak bugs.** Landslide records the state of the heap before the test case begins, and compares it to the state of the heap after the test case ends. If memory allocated during the test was not freed, Landslide assumes that it was leaked. (Some kernel designs may legitimately behave this way, so this bug-check may be disabled when testing such kernels.)[7]

5. **Infinite loop bugs.** Landslide judges whether the kernel has entered an infinite loop by comparing the current branch of the decision tree to past executions of the same test case. Figure 5.3 depicts such tree structures in abstract.

   - **Infinite loops without decision points.** While exploring the decision tree, Landslide computes the average number of instructions executed between two consecutive decision points. If at any point the current number of instructions executed since the most recent decision point exceeds this average times a constant factor (arbitrarily chosen to be 2000), Landslide assumes the kernel must have gotten stuck in an infinite loop.
   - **Infinite loops around decision points.** Landslide also computes the average number of decision points in each branch of the decision tree (the average "branch depth"). If the depth of the current branch ever exceeds this average times a constant factor (arbitrarily chosen to be 20), Landslide assumes the kernel must have gotten stuck in an infinite loop.

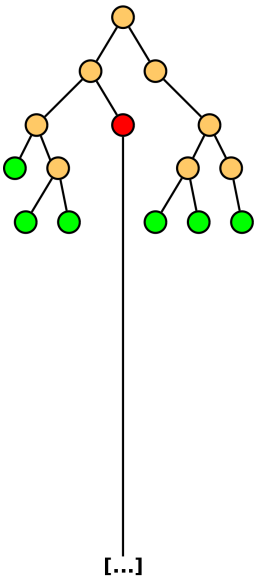
   In our experience, these heuristics have never falsely identified an infinite loop when the kernel was making real progress. However, both of these heuristics require a minimum number of branches to be explored before Landslide considers the already-explored tree structure to be statistically significant (arbitrarily chosen to be 20). As such, these heuristics occasionally fail to trigger on a real infinite loop in cases when the loop occurs before enough "safe" branches were explored. In the future we might improve this heuristic by scaling the associated cutoff factor depending on how many branches were explored: the

[5] In the same way, Landslide also detects calls to `free` on blocks that were already freed or never allocated at all.

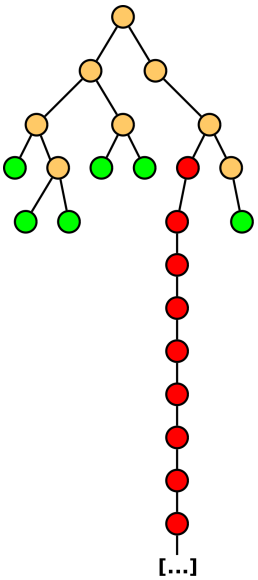
[6] Except for idle, if it exists.

[7] There is, of course, much room for improvement in this metric, but it is not part of the research contribution.

fewer branches explored so far, the less reliable the comparison, and hence the higher the cutoff factor to be used.



(a) Infinite loop without decision points. Disproportionately many instructions have been executed since the red state, so the kernel is probably stuck in a "tight" loop.



(b) Infinite loop around decision points. Disproportionately many decision points have been encountered on the red branch, so the kernel is probably stuck in a loop around certain decision points.

Figure 5.3: Landslide judges whether the kernel has gotten stuck by analysing the structure of the execution tree. The red branches indicate non-terminating thread interleavings, which Landslide would identify by comparing them with other branches in their respective trees.

5.4 Partial-Order Reduction

We make use of Dynamic Partial-Order Reduction (DPOR), the state-space pruning algorithm presented in [FG05]. DPOR requires two sets to be computed that describe the concurrency relationship between transitions: the happens-before relation, and the memory independence relation. Here we discuss the specifics of implementing these in our environment.

5.4.1 Happens-Before Relation

The happens-before relation expresses for each pair of transitions whether executing the first one is required to "enable" the second one. In order to establish the relation, Landslide uses the state of the scheduler runqueues (Section 5.2.2), which are snapshotted at every decision point (Section 5.2.5).

Unfortunately, a thread's presence or absence on the scheduler runqueue does not necessarily correspond to whether it is *runnable* at any point. We identify three exceptions, as foreshadowed in Section 3.3:

- **Current thread not on runqueue.** If the currently-running thread is not stored on the runqueue, we identify it as runnable anyway (except in the special case of idle, as described below).
- **Idle thread.** Some kernels may have an explicit idle thread and store it on the runqueue, with explicit code to skip over it if other threads are runnable. Hence, if the idle thread is on the runqueue (and/or the current thread), it is only runnable if no other threads are runnable.
- **Yielding mutexes.** In kernels whose mutexes (or other synchronisation idioms) leave threads on the runqueue when they are notionally blocked, Landslide classifies such threads as not runnable.

Using this notion of runnability, the happens-before relation is established as follows. Let X be a transition and Y be a subsequent transition in a branch that was just explored, and T_Y be the thread associated with Y. We say that X *enables* Y if T_Y was not runnable immediately before X, but was runnable immediately after. Then, a transition A happens-before a transition B if T_A is the same thread as T_B, if A enables B, or if A happens-before some transition C that enables B.

5.4.2 Memory Independence Relation

The independence relation expresses which transitions do not read-and-write or write-and-write to the same shared memory addresses. Landslide computes this relation using the set of shared memory accesses described in Section 5.2.3. We encountered two challenges pertaining to memory independence:

- **Always-accessed memory locations.** In kernel-space, every thread switch goes through common scheduler and context-switcher routines. These routines inevitably access scheduler data structures, such as the runqueue. The timer tick counter is another example: this global variable is not necessarily accessed every transition, because voluntary reschedules do not involve timer interrupts, but every involuntary preemption will cause a write to this counter.

  If we include such memory accesses in the independence relation, it will result that all transitions conflict, and DPOR won't be able to achieve any reduction. To allow for state space reduction, we sacrifice our ability to find races involving these particular accesses by ignoring them. Currently we require the user to identify these locations as part of the instrumentation process, as described in Section 6.2.2.

- **Freed memory poisoning.** A use-after-free bug happens whenever one thread accesses an address within an allocated block that another thread previously freed. Even if no other code makes a conflicting access to the same address after the `free`, accessing dynamically-allocated memory after it has been freed is illegal no matter what.

  Hence, even if the second thread that freed the block never accesses the particular address that the first thread used, the second thread's `free` still logically conflicts with the first thread's access. Landslide addresses this by treating every call to `free` as a *write* access to every address within the freed block when computing shared memory conflicts.

5.4.3 Soundness

Because we recommend using Landslide with coarse-grained decision sets, as opposed to mandating a decision point between every pair of shared memory accesses, it is possible for Landslide to overlook race conditions that require finer-grained interleavings to expose. In this way, Landslide's search is not "sound", but instead false-negative-oriented.

We do, however, claim that Landslide's search is sound in a different way: if the provided decision points are sufficient to express an interleaving that would expose a race, then Landslide will find it. That is, if Landslide is exploring an execution tree with bugs in some branches, it will eventually find a bug.

Dynamic Partial Order Reduction works by identifying "evil ancestors" for each transition in a branch after executing that branch. Intuitively, a transition's evil ancestor is another transition such that executing the test with the order of those two transitions reversed could cause different concurrent behaviour to arise. [FG05] and [SBGH11] present a partial order reduction algorithm in which only the first evil ancestor of each transition need be considered when identifying which alternate interleavings need to be explored.

We found that implementing DPOR in this way violated our second notion of soundness, because it relies on the invariant that each transition contains only one inter-thread communication event (in kernel-space, this means a shared memory access). However, a key point of Landslide's recommended usage pattern is to sacrifice maximally-fine-grained decision points for performance, and so multiple different shared memory conflicts may happen in each transition. Figure 5.4 demonstrates a counterexample to the soundness of identifying only one evil ancestor per transition given our usage pattern.

Instead, Landslide considers every evil ancestor of each transition when performing DPOR. This modification enables Landslide to avoid the problem presented in the figure, because it will identify both alternate interleavings after exploring the first one. We claim without certainty that this change restores the second notion of soundness to Landslide's search, and leave proving this property to future work.

	Thread 1	Thread 2	Thread 3
A	`x=42;`		
B		`y=31337;`	
C_1			`if (y==31337) assert(x==42);`

(a) Race-free interleaving.

	Thread 1	Thread 2	Thread 3
A	`x=42;`		
C_2			`if (y==31337)`
B		`y=31337;`	

(b) Alternate interleaving explored with DPOR.

	Thread 1	Thread 2	Thread 3
B		`y=31337;`	
C_3			`if (y==31337) assert(x==42);`
A	`x=42;`		

(c) Buggy interleaving.

Figure 5.4: This sample code defeats a DPOR implementation that uses only the first evil ancestor if a decision point is not defined between the two lines of thread 3's code. The first interleaving is explored first; DPOR identifies transition B as the first evil ancestor of C_1, and runs the second interleaving next, reordering them. In the second interleaving, A is no longer a second evil ancestor of C_2, so the third interleaving is never explored, and the assertion failure is missed.

5.5 Debugging Feedback

In its unique position of control over when the kernel gets preempted and which thread gets scheduled at each context switch, Landslide has the capability to provide the user with detailed information about a test case's execution. When Landslide determines that a bug was found, it immediately aborts exploration of the decision tree, and prints a *decision trace*: a comprehensive report of the particular interleaving of thread transitions that caused the bug to appear. Figure 5.5 depicts a simple decision trace for a use-after-free bug.

The decision trace explains each preemption or voluntary reschedule in the interleaving: which thread used to be running, which thread was chosen to run instead, and the stack trace of the former thread at the point from which it was switched away.

Landslide also prints the stack trace of the currently-running thread at the point where the bug was found, and (optionally) drops the user into the Simics debugging prompt. Depending on the nature of the bug found, Landslide also provides more detailed information:

1. If the kernel has panicked, Landslide prints the message used in the panic/assertion.
2. If a use-after-free bug is found, Landslide prints information about the most-recently-freed chunk containing the accessed address: the stack trace and TID both for when it was allocated and freed.
3. If deadlock is detected, Landslide prints the cycle of TIDs that are blocked on each other.[8]

[8]This currently works only when using the annotations for yielding mutexes, though it is not difficult to

```
[SCHEDULE]      thread 4 vanished
[SCHEDULE]      switched threads 4 -> 3
[MEMORY]        USE AFTER FREE - read from 0x0015a8f0 at eip 0x00104209
[MEMORY]        Heap contents: {...}
[MEMORY]        [0x15a8f0 | 4136] was allocated by TID3 at (...)
[MEMORY]                                and freed by TID4 at (...)
[BUG!]          ****     A bug was found!    ****
[BUG!]          **** Decision trace follows. ****
[BUG!]          1:  1347079 instructions, old 3 new 4, current 4
[BUG!]              TID3 at 0x00105a10 in context_switch,
[BUG!]                      0x001041f4 in thread_fork,
[BUG!]                      0x0010362b in thread_fork_wrapper
[BUG!]          2:  1350725 instructions, old 4 new 3, current 3
[BUG!]              TID4 at 0x00105a10 in context_switch,
[BUG!]                      0x00104681 in yield,
[BUG!]                      0x00104570 in vanish,
[BUG!]                      0x00103708 in vanish_wrapper
[BUG!]          Stack: TID3 at 0x00104209 in thread_fork,
[BUG!]                         0x0010362b in thread_fork_wrapper
[BUG!]          Total decision points 24, total backtracks 5
[BUG!]          Average instrs/decision 16155, average branch depth 5
```

Figure 5.5: A sample decision trace (with extraneous text trimmed) that Landslide generated using the `double_thread_fork` test case (Section 6.4).

4. If a memory leak is suspected, Landslide prints how many bytes bigger the heap is after the test ended than when the test began.[9]

5. If an infinite loop is suspected, depending on which heuristic was triggered, Landslide prints either the number of instructions since the last decision point (and the previous average) or the current branch depth (and the previous average).

Landslide can also provide other useful information, even in cases where it did not find bugs, to help with the user's process of configuring decision points. Instead of exploring alternative interleavings, it can stop execution after the first interleaving and print out the set of decision points that were identified, along with the shared memory conflicts and happens-before relations for pairs of transitions between the decision points.

implement more generally.

[9]Future feature: Printing when-allocated stack traces for each suspicious heap chunk.

Chapter 6

Using Landslide

In this chapter we discuss how to use Landslide with a kernel that meets 15-410's Pebbles specification. We list some design requirements that a Pebbles kernel must meet, document the instrumentation process and how to customise Landslide's search behaviour, and tell how to interpret Landslide's results.

In preparation for conducting the student user study (Section 7.1), we distributed a user guide roughly similar to this chapter.[1]

The recommended "attack plan" for using Landslide corresponds directly to the sections in this chapter. Section 6.1 lists the requirements a kernel must fulfill to be compatible with Landslide. Section 6.2 describes the process of instrumenting a kernel, which involves annotating the kernel, filling out a configuration file, and instrumenting within Landslide itself. Section 6.3 describes ways to configure Landslide's behaviour to help focus its search for bugs.

6.1 Kernel Requirements

In addition to meeting the Pebbles specifcation for 15-410, there are some additional restrictions that Landslide imposes, some inherent to its testing model, and others as artifacts of its interface/implementation.

6.1.1 Scheduler Functionality

In order to cause desired preemptions, Landslide needs to assume that the core of the scheduler works. In short, this means that timer interrupts will trigger context switches, and that if a thread is "runnable", at any point where interrupts/preemption is enabled, a finite number of timer preemptions will eventually cause that thread to run.

[1]The user guide itself is available at `http://bblum.net/landslide-guide.pdf`, and at `http://www.contrib.andrew.cmu.edu/~bblum/landslide-guide.pdf`.

For deadlock detection, if the kernel has mutexes which loop around a call to `yield` (or similar), they must be annotated as described in section 6.2.1. If mutexes explicitly deschedule blocking threads, no special annotations are needed, because the other annotations do the job.

Landslide does not support kernels that spin-wait anywhere, whether in mutexes or otherwise, especially when waiting for keyboard input or in `sleep`. Relatedly, the kernel must never run the idle loop when not truly idle, because Landslide uses this as a way of telling when all threads are wedged and/or when the test is done running.

6.1.2 Virtual Memory

Landslide does not have much to do with the virtual memory system, but the kernel must direct-map most of kernel memory, including the heap, globals, and kernel thread stacks. Landslide needs this to read values out of memory and to identify memory conflicts.[2]

The kernel should also use LMM (the `malloc` package in the 15-410 base code) for dynamic allocation, to avoid having to manually instrument a custom allocator.

6.1.3 System Calls

In order to run with Landslide, the kernel must be able to boot up to the shell prompt,[3] receive keyboard input to make the shell start a test case, and return to the shell prompt after the test case finishes. The kernel must have the following system calls implemented and working in at least a rudimentary way: `fork`, `exec`, `vanish`, `wait`, `readline`.

The following system calls are exercised in some, but not all, of the test cases we distribute, so are helpful to have, but not necessary: `thread_fork`, `yield`.

Apart from the system calls which init and shell execute, the rest are irrelevant and do not need to be implemented to use Landslide on a Pebbles kernel.

6.2 Instrumenting Kernels with Landslide

Here we document the interface by which Landslide understands the execution of the guest kernel. Landslide tries to be as design-agnostic as possible, after assuming that the kernel implements the Pebbles specification, but it still needs to know about the implementation of certain abstractions within the kernel.

```
int thread_fork()
{
    thread_t *child = construct_new_thread();
    tell_landslide_forking();
    add_to_runqueue(child);
    tell_landslide_decide(); /* Interrupt me here!  */
    return child->tid;
}
```

Figure 6.1: Example annotated code. In theory, the use of `tell_landslide_decide()` in the place where a preemption needs to occur is not necessary for Landslide to know to preempt then; it is shown for the sake of demonstration.

6.2.1 In-Kernel Annotations

We provide a set of functions that the kernel needs to call at certain points during its execution to communicate what interesting events are happening. Figure 6.1 shows some example code annotated for Landslide.

Important Note about Runqueues

Landslide uses two of these annotations, `on_rq()` and `off_rq()`, for tracking the state of the guest kernel's scheduler runqueue at each point during execution. Note that this refers to the *actual runqueue data structure*, which does not necessarily correspond to the abstract "set of all runnable threads", depending on whether or not the kernel keeps the current thread on the runqueue or off of it.

If the kernel does not keep the currently-running thread on the runqueue, the user must also use `kern_current_extra_runnable` (Section 6.2.3).

Required Annotations

Some of these annotations may need to be used in "sensitive" regions of scheduler code, where it may not necessarily be clear at what point during the operation the annotation should go. Landslide assumes the scheduler already protects these regions by disabling preemption, and so the exact position of the annotation does not matter as long as it is within the preemption-disabled range.

- `tell_landslide_thread_switch(int new_tid)`: Call this in the context switcher, when a new thread is switched to.

[2]It would be simple to support more advanced virtual memory set-ups, in which the same virtual address has different physical memory mappings at different times, but this is not implemented.

[3]In Pebbles, the bootup sequence is usually fairly simple, involving the init process spawning the shell.

- `tell_landslide_sched_init_done()`: Call this when the scheduler is done being initialized. Until this is called, Landslide will ignore all other annotations.

- `tell_landslide_forking()`: Call this when a new thread is being forked. (Hint: call it twice, once in `fork` and once in `thread_fork`.)

  Call it "just before" the action which makes the new thread runnable. It does not matter whether the kernel just adds it to the runqueue (in which case the next annotation called would be `thread_on_rq`) or begins running it immediately (in which case `thread_switch`); Landslide handles both cases.

- `tell_landslide_vanishing()`: Call this when a thread is about to go away. Call it "just before" the relevant context switch; i.e., the one that should never return.

- `tell_landslide_sleeping()`: Call this when a thread is about to go to sleep (that's `sleep()` the system call). Call it "just before" the relevant context switch; i.e., the one that won't return for as long as the thread requested to sleep.

- `tell_landslide_thread_on_rq(int tid)`: Call this when a thread is about to be added to the runqueue. (Make sure this call is done within whatever protection is also used for the runqueue modification itself.)

- `tell_landslide_thread_off_rq(int tid)`: Call this when a thread is about to be removed from the runqueue. (Same protection clause as above applies.)

- Mutex annotations. These are only necessary if the kernel uses mutexes that leave blocked threads on the runqueue, rather than explicitly descheduling them.

  - `tell_landslide_mutex_locking(void *mutex_addr)`: Call this when a thread is about to start locking a mutex. Landslide needs the mutex address for deadlock detection, to track which mutex each thread is blocked on.
  - `tell_landslide_mutex_locking_done()`: Call this when a thread finishes locking a mutex (and now owns it).
  - `tell_landslide_mutex_unlocking(void *mutex_addr)`: Call this when a thread is about to start unlocking a mutex.
  - `tell_landslide_mutex_unlocking_done()`: Call this when a thread finishes unlocking a mutex (and no longer owns it).
  - `tell_landslide_mutex_blocking(int owner_tid)`: Call this when a thread is contending on a locked mutex, and hence is no longer logically "runnable" yet still on the runqueue. Landslide needs to know who owns the mutex for deadlock detection.

Optional Annotations

- `tell_landslide_decide()`: Call this to define additional decision points. See section 6.3.1.

- `assert()` or `panic()` - The more important invariants for which a kernel has `asserts`, the more likely Landslide is to find bugs that would trigger them; otherwise, even if they do happen, Landslide might never know (just like in conventional stress testing).

6.2.2 Configuration File

There is also a config file the user needs to fill out, called `config.landslide`. Some of the fields are required information, and some are tweaks that change the way Landslide behaves. We omit discussion of the required fields, and explain the optimisations and behaviour tweaks that Landslide supports using this file in Section 6.3.2.

6.2.3 Instrumenting Within Landslide

There are two functions that the user needs to implement in Landslide itself, in a file called `student.c`, to express certain kernel designs that might be more complicated than a simple true/false condition or integer.

- `bool kern_current_extra_runnable(conf_object_t *cpu)`: See Section 6.2.1. This says whether the current thread is "logically runnable" despite not being on the runqueue itself (that is, `tell_landslide_on_rq` won't have been called for it).

- `bool kern_ready_for_timer_interrupt(conf_object_t *cpu)`: This function can be used to express when the scheduler is "locked" in a way not involving disabled interrupts. Landslide will avoid trying to preempt the kernel whenever this is false.

Both of these functions are predicates on the guest kernel's execution state, usually on the contents of its scheduler's data structures. Example implementations are given in Figure 6.2.

```
bool kern_current_extra_runnable(conf_object_t *cpu)
{
    int tcb = READ_MEMORY(cpu, GUEST_CURRENT_TCB_POINTER);
    int state_flag = READ_MEMORY(cpu, tcb + GUEST_TCB_STATE_FLAG_OFFSET);
    return state_flag == GUEST_TCB_RUNNABLE;
}
bool kern_ready_for_timer_interrupt(conf_object_t *cpu)
{
    int x = READ_MEMORY(cpu, GUEST_PREEMPTION_FLAG);
    return x == GUEST_PREEMPTION_ENABLED;
}
```

Figure 6.2: Example implementations of the two functions in `student.c`.

6.3 Configuring Landslide's Behaviour

6.3.1 Decision Points

Getting a good set of decision points is important to being able both to explore reasonably quickly and to produce meaningful interleavings that are likely to find bugs. After getting Landslide working with the default set of decision points (voluntary reschedules only), the next step is to add some more. We list here the general steps in this process.

- **Indicating decision points.** Insert calls to `tell_landslide_decide()` in the guest kernel where Landslide should identify decision points. We recommend at the start of `mutex_lock` and at the end of `mutex_unlock`, though it is also important to use one's own intuition, depending on which system call(s) are being tested.
- **Ignore always-conflicting memory locations.** Use `sched_func` and `ignore_sym` to identify shared memory locations that Landslide should ignore, to enable better pruning.
- **Examine decision points.** Set `DECISION_INFO_ONLY` to make Landslide print the decision points instead of exploring. Examine the decision set, and figure out which ones are irrelevant to the test.
- **Focus decision points.** Use `within_func` and `without_func` to specify which components of the kernel Landslide should identify decision points in, to reduce the state space and focus only on relevant interleavings.

6.3.2 Search Parameters

Landslide's configuration file supports several options for changing the way it explores the decision tree.

Optimizations

These configuration options help Landslide identify potentially-conflicting shared memory accesses that it should ignore (Section 5.4.2).

- `sched_func`: When functions that make up the kernel's scheduler are identified, Landslide knows to ignore shared memory accesses from them. Accesses to scheduler data structures happen during *every* transition, so ignoring these is necessary for achieving any reduction at all.
- `ignore_sym`: The user may wish to ignore memory accesses from other global data structures as well. For example, ignoring a mutex which is locked very frequently, but irrelevant to the actual test case, will result in an independence relation that enables much better state space reduction without losing the ability to find bugs in the parts of the kernel the user is interested in testing.

These configuration options enable the user to whitelist or blacklist certain components of the kernel for the purposes of identifying decision points.

- `within_function`: If the user configures decision points to happen in common code paths, such as `mutex_lock`, it will likely generate more branching than needed. For example, if testing `vanish`, the user may judge there to be no need to branch on a `mutex_lock` in `exec`.

  This configuration option allows the user to whitelist functions so that Landslide will only decide if the current thread is "within" one of those functions.

- `without_function`: Similar to above, but a blacklist. These two can be used together; later invocations take precedence over earlier ones.

Behaviour Tweaks

- `EXPLORE_BACKWARDS`: In which order to explore the branches? Backwards means with more preemptions first; forwards means tending to let the current thread keep running more often. Backwards tends to find bugs more quickly, but produces longer decision traces. (Section 8.7.1)

- `DECISION_INFO_ONLY`: This option makes Landslide stop after one branch, and output the list of decision points it identified using the current config. Useful to see all decision points, and tweak settings to get rid of frivolous ones.

6.4 Test Cases

We ship Landslide with a suite of small test cases designed to expose several common races that students frequently encounter during 15-410 Project 3. We list their code in Appendix A.

- `vanish_vanish`: Tests when a parent and child process `vanish()` simultaneously.

- `double_wait`: Tests interactions of multiple waiters on a single child.

- `double_thread_fork`: Tests for interactions of multiple threads in one process vanishing.

- `yield_vanish`: Tests for interactions between `yield()` and `vanish()`.

6.4.1 Landslide-Friendly Test Characteristics

It is important to note why all of these test cases are "Landslide-friendly": they all perform very little work on a single run, enabling Landslide to completely explore the state-space. They also run several, but not too many, threads at once, producing potentially interesting interleavings.

Chapter 7

Evaluation

In evaluating Landslide, we sought to answer two overall questions.

- Is Landslide an effective use of time by a developer seeking to find and understand race conditions in their code? How can Landslide be improved to improve the time tradeoff?
- When testing kernel code with a systematic testing framework, what overall use patterns are most effective for getting meaningful results? How can future work employ such patterns to build more sophisticated systematic testing tools for kernels?

To answer these questions, we evaluated Landslide in two ways. First, we conducted a user study with students of 15-410 during the spring 2012 semester, to understand the ease and difficulty of Landslide's user interface process, to understand how to polish it into a more useful debugging tool. Second, we did a case study of six bugs in two kernels, investigating them in-depth to understand how systematic testing might best be harnessed to find bugs without in advance knowing the decision set necessary to expose them.

7.1 User Experience

To evaluate the experience of non-expert users working with Landslide, we met with 5 groups of students in 15-410 (9 students in total) during the final week of the kernel project in spring 2012, and had them use Landslide with their own kernels. Additionally, one former 15-410 Teaching Assistant (TA) participated in the study.

In order to garner student interest in Landslide, the author gave a lecture in 15-410 on systematic exploration in general and on Landslide specifically during the second-to-last week of the kernel project.[1]

[1]The lecture slides are available at `http://www.cs.cmu.edu/~410-s12/lectures/L30_Landslide.pdf`, and at `http://bblum.net/landslide-lecture.pdf`, and at `http://www.contrib.andrew.cmu.edu/~bblum/landslide-lecture.pdf`.

We wished to measure which phases of using Landslide are most time-consuming, so we asked students in intervals between 30 and 60 minutes to record the amount of time they had spent doing each phase:

1. Annotating their kernel (Section 6.2.1).
2. Implementing instrumentation within Landslide (Section 6.2.3).
3. Fixing problems encountered getting Landslide to run, such as revising their kernel to meet Landslide's requirements and debugging incorrect instrumentation.
4. Customising Landslide's search parameters such as decision points (Section 6.3).
5. Analysing the output from bugs that Landslide found.

We also wanted descriptive feedback, to evaluate which specific parts of the process of using Landslide need improved. We asked students to write brief remarks whenever they recorded amounts of time spent, and also after they were finished to answer the following questions.

- Were you able to get Landslide to work with your kernel (i.e., run through a minimal exploration tree completely)? If so, how long did it take, from sitting down to getting it to work? If not, did your kernel do something that was incompatible with Landslide, or were you unable to get the instrumentation right for some other reason?
- Did you find bugs while using Landslide, that you imagine would have been very hard to find otherwise? Did Landslide's output help you understand what caused them?
- Were you able to get Landslide to say "you survived!" with custom decision points? Did you think the set of decision points you used for this provide a strong guarantee about the absence of races?
- Describe your experience configuring the set of decision points. What was intuitive, obvious to do? Did you feel stuck at any point, not knowing where to go next?
- Was there anything you wanted to make Landslide do that it didn't support?

Finally, we briefly studied each bug that the students found: whether they were deterministic or race conditions, whether they were deep unsolved concurrency problems or simple oversights, and the complexity of the necessary fixes.

Of five student groups that we worked with during the user study, four groups completed the instrumentation process and got Landslide to explore at least a minimal decision tree. (A minimal decision tree is one resulting from decision points only on voluntary reschedules, which Landslide automatically identifies.)

Group	Minutes spent doing...						Total time spent	
	annotating	config	student.c	fixing	customising	found races	required	refinement
Group 1	35	–	–	5	–	–	–	–
	5	30	10	100	–	–	145	–
Group 2	20	30	10	80	–	10	140	10
	15	45	30	60	–	30	150	30
Group 3	15	10	5	60	30	30	90	60
	15	10	5	60	30	30	90	60
Group 4	10	42	3	5	30	–	60	30
	15	35	–	5	35	–	–	35
Group 5	50	15	–	15	–	–	–	–
Former TA	40	20	3	95	–	30	158	30
Average	**22**	**26.33**	**9.43**	**48.5**	**31.25**	**26**	**119**	**36.43**

Table 7.1: Self-reported times spent by users on each phase while using Landslide. "–" indicates a student reported spending no time on a particular phase. "Required" indicates the sum of the times from the first four phases; "refinement" inducates the sum from the last two phases. "–" indicates that a user did not report spending any time on a phase; the "required" field is "–" if any of its represented phases are, and "–" fields are not reflected in the averages.

7.1.1 Time Breakdown

While the students were using Landslide, we asked them at intervals from 30 to 60 minutes to record how much time they had spent on each phase of instrumenting and using Landslide.

- **Required instrumentation.** These phases represent work necessary to getting Landslide to work with a kernel at all.
  - **Annotating the kernel.** In this phase, the student modifies their kernel to use the provided annotations (Section 6.2.1).
  - **Writing the configuration file.** In this phase, the student fills out certain information about their kernel in `config.landslide` (Section 6.2.2).
  - **Instrumenting within Landslide.** In this phase, the student implements the two instrumentation functions in `student.c` (Section 6.2.3).
  - **Fixing instrumentation problems/crashing.** In this phase, the user repairs problems that arose due to incorrect instrumentation (or bugs in Landslide itself, in some cases).
- **Extra refinement.** These phases represent additional work to be done after successfully exploring a minimal decision tree. When such minimal exploration finds no bugs, such additional work is necessary to provide meaningful results.

– **Customising decision points.** In this phase, the user configures Landslide to use decision sets more refined than the minimal one, in search of hard-to-find bugs (Section 6.3).

– **Investigating found race conditions.** In this phase, the user attempts to understand and fix bugs that Landslide found and reported.

Table 7.1 presents the time breakdown among phases that students experienced while using Landslide, visualised in Figure 7.1. On average, students spent two hours getting Landslide to work with their kernels, and slightly more than half an hour customising the search and/or studying found bugs on top of that.

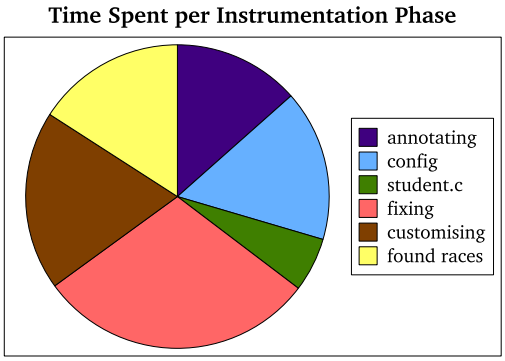


Figure 7.1: Average time breakdown for each phase of instrumenting a kernel with Landslide, based on student self-reported times.

We consider the absolute values of these numbers largely artifacts of the user interface's quality (i.e., they could be improved with extra interface polishing, though they also currently present an upper bound). The relative values among phases, however, serve to give a notion of which phases were most difficult.

We do not think it useful to consider these times amortised over how many bugs were found for each group, because in general we do not know whether each group would have found more bugs with marginal extra effort, having already paid the initial cost of instrumenting. (The vanish/vanish bug we describe finding ourselves in Section 7.1.3 serves as an example of this perennial possibility.) Instead, we claim that the fact that Landslide found bugs for all groups who completed the required instrumentation proves its baseline effectivness as a debugging tool, and we note that the time tradeoff will become more worthwhile the more the user interface improves.

7.1.2 Descriptive Feedback

The students provided specific remarks which fell into two categories: issues with Landslide's user interface, and needing to change something in one's kernel to make it work with Landslide.

User Interface Feedback

For Landslide to be appealing as a debugging tool, we should improve the user interface (or the infrastructure, in some cases) to address the following issues.

- **Difficulty of debugging incorrect instrumentation.** When the user-provided annotations/instrumentation are incorrect, Landslide's error messages are difficult to understand, and often don't point to the real problem.

- **Interpreting debugging output.** Some students reported the decision trace representation could be better. One student reported the output was easy to follow by "tracing the life of the buggy thread", which is an insight into the learning process we should capitalise on.

- **Ease of configuring decision sets.** Most students seemed to find the process of adding more decision points intuitive. One reported, "quick and effective at detecting basic bugs/races." While this interface could always be improved, of all the parts of the user experience, this process seems to need it least.

- **General feature wishlist.** The following features, of varying difficulty to implement, would have improved user experience overall.

  - Being able to continue exploration after interrupting Simics to work with the debug prompt, instead of having to start over.
  - Support for multiprocessor kernels.
  - Cutting down overall simulation time.

Technical Feedback

Students also described compatibility issues they faced between their kernels and Landslide, which required them to make certain changes to their kernels.

- **Special-case context switch behaviour.** One kernel had a special context-switcher for the case when a thread was vanishing, which had to be special-cased in the instrumentation. Some kernels had different context switch designs for just-forked threads (as discussed in the "Thread Creation" bullet in Section 3.3), and Landslide's code had to be modified during the study to support different designs.

- **Unexpected compiler optimisations.** Listing scheduler functions in `config.landslide` could be confusing if the compiler inlined them.

- **Driver thread interfering with test lifecycle tracking.** One kernel (Pathos, the 15-410 reference kernel) had to be modified to ensure the keyboard thread ran once before the test started, which was not mandated by the kernel design. (If the thread didn't run, it would conflict with the population-tracking described in Section 5.2.6, and Landslide would not be able to detect when the test was over.)

- **Kernel behaviour requirement violations.** The majority of groups needed to change their scheduling around the idle thread in order to meet the requirement that the kernel never run idle when progress could otherwise be made (Sections 3.3 and 6.1.1). Also, one kernel had to have its custom memory allocator disabled, because Landslide only supports LMM, the allocator that 15-410 provides.

  The idle requirement is an inherent part of Landslide's model, whereas the dynamic allocator requirement could be replaced with a richer set of optional annotations.

Additionally, two enterprising users found ways to avoid implementing the functions in `student.c` (Section 6.2.3). It is worth considering integrating these approaches into the recommended approach to eliminate the need for `student.c` entirely.

- One student bypassed the `kern_ready_for_timer_interrupt` annotation by disabling their scheduler's preemption-disabling mechanism and replacing it with explicit disabling of interrupts. This caused the kernel to context switch every timer interrupt, so Landslide did not need to be aware of extra conditions that would cause the scheduler to ignore timer ticks.

- The former TA was able to avoid the `kern_current_extra_runnable` annotation by using the `on_rq` and `off_rq` annotations (Section 6.2.1) to express the "abstract set of runnable threads" (which included the currently-running thread, even though the runqueue didn't) instead of the literal runqueue itself. We did not advertise this method because we suspected students might make more mistakes instrumenting this way, but it might be worth advocating anyway for the sake of making the interface simpler.

7.1.3 Landslide Victories

Of the four groups that completed the required instrumentation, two found race conditions in their kernel. Additionally, Landslide helped all four groups find deterministic bugs: three groups had previously unknown problems with running the idle thread inappropriately, and the other group had a simple use-after-free.

We list how much progress each group made, and whether or not they found any bugs.

1. Group 1 got Landslide to explore a minimal decision tree, and also explored `double_wait` and `vanish_vanish` with several combinations of different decision points around `mutex_lock` and `mutex_unlock`. They found one race and a problem with idle. After fixing these bugs, they were able to completely explore both test cases with decision points on both `mutex_lock` and `mutex_unlock`, which we consider a strong negative result (i.e., no bug found in those two test cases with fine-grained interleavings).

2. Group 2 got Landslide to explore a minimal decision tree. They did not configure any extra decision points. They found a problem with idle, and after fixing it, found a vanish-related race condition in the fix. They also found a bug in an assertion which they had

added during the process, but that bug was deterministic, so we consider this part of the "fixing incorrect instrumentation" process.

3. Group 3 got Landslide to explore a minimal decision tree, and also configured extra decision points for `vanish_vanish`. They found only a deterministic use-after-free bug.

4. Group 4 got Landslide to explore a minimal decision tree. They found only a problem with idle.

5. Group 5 did not invest enough time to get Landslide to explore a minimal decision tree.

Race Conditions Found

Two groups found race conditions using Landslide. We describe them here.

- **Too many waiters allowed.** Using the `double_wait` test case, Group 1 found a bug in which more threads invoking `wait` would be allowed to block than the number of child processes that could ever be reaped, and some of the waiters would end up blocking forever. The fix for this was two lines of code. The group claimed it could have been found with an "in-depth" code review, though we claim Landslide's contribution here was still non-trivial, because (as exhibited here) such bugs exist anyway for lack of sufficient review.

- **Accidental return to a vanished thread.** Group 2 had a problem with the scheduler running idle when it was not supposed to; when they fixed it, however, their implementation would return to a thread that had already vanished in the case when no thread was left to run. It is unclear if this race would have been exposed without Landslide. Fixing it required adding an extra condition to the code that checked whether the kernel should idle.

We also note that, though Group 2 stopped using Landslide after getting it to explore the minimal decision tree, if they had configured decision points around `mutex_lock` (which was the recommended next step in the process), they would have found a subtle race in the reparenting section of their `vanish` code using the `vanish_vanish` test. We verified this by having an ex-TA re-instrument their kernel and add the decision points: from the time of exploring a minimal decision tree to finding the bug with the new decision set took half an hour of work, and fully understanding the nature of the bug took an hour on top of that.

This bug is similar to the LudicrOS vanish/vanish bug studied in Section 7.2, and would have required a complicated redesign of the `vanish` implementation to fix.

We consider this a mixed blessing: Landslide was capable of finding a deep concurrency bug in a student kernel, but the instrumentation process was expensive enough that they were not motivated to continue working, and stopped just short. We believe that improving Landslide's interface (Section 8.1) will improve the success rate in such cases.

Deterministic Bugs Found

We also claim that Landslide is useful in its ability to find certain types of non-concurrency-related bugs. Though its primary purpose is to find race conditions, we found that it caught two main types of bugs that the students were previously unaware their kernels had.

- **Inappropriate idling.** Two groups had errors in their keyboard handling code, where when a newline was received when the kernel was idle, a thread blocked on `readline` would be added to the runqueue and but not switched directly to, so the kernel would continue idling even though there was useful work to do. Another group left idle on the scheduler runqueue, and had no special-case code to skip over it, so would interleave idle arbitrarily with other threads. We consider both of these performance problems.

- **Deterministic use-after-free.** One group had a deterministic use-after-free bug in their `wait` implementation, in which they freed something and immediately dereferenced it. This hadn't presented a problem because the block was never reallocated during the critical window, which would have taken a complicated and fine-grained interleaving, but Landslide's heap checker detected the use-after-free immediately, without the need for any interleaving.

7.2 Bug Case Studies

While building Landslide, we worked primarily with two kernels: one, the kernel the author wrote as a student in 15-410 (Fall 2008), and another, a kernel the author graded as a TA for 15-410 (Fall 2011). We refer to the first kernel as "POBBLES" and to the second, with permission, as "LudicrOS".

With Landslide, we found four bugs in POBBLES and two bugs in LudicrOS, and investigated them more deeply.

7.2.1 Bug Descriptions

Here we describe each of the six bugs that we used as case studies. We rate their complexity and the estimated ease of fixing them.

POBBLES vanish/vanish (a)

We introduced this bug into POBBLES's vanish implementation. When a parent and child invoke vanish concurrently, the parent needs to modify the child's parent pointer to reparent it to the init process, and the child needs to read its parent pointer to signal to its parent that it can be reaped.

In version (a) of this bug, we introduce a mutex for each process's parent pointer. While holding the lock around all related process structure accesses prevents data races, the circular nature of the lock acquisitions between parent and child results in deadlock.

This bug is exposed with the `vanish_vanish` test case. It requires a very specific interleaving sequence to uncover. The fix is nontrivial, because the circular data structure access still needs to happen, but be protected in a way that doesn't involve circular wait. Vanish races are notorious among 15-410 course staff as being difficult to find with the naked eye, and also difficult to figure out how to fix.

POBBLES vanish/vanish (b)

We also introduced this bug into POBBLES's vanish. Version (b) is similar to version (a), except we removed the parent pointer mutex entirely. This results in a data race around the parent pointer, in which the child can end up accessing the wrong parent either before or after it is actually moved to init's child-list.

This bug is exposed with the `vanish_vanish` test case. The complexity and difficulty of fixing are the same as in version (a).

POBBLES wait/wait

Finding this bug was somewhat of a surprise. When two threads in the same process wait for a child to exit, they both block on the same condition variable, and one of them has a pointer to the other in its `nobe->cond_queue_next` pointer. However, when awakened, the condition variable code did not clear the value of this pointer. Fortunately, an assert statement in the thread destruction code caught that the pointer was non-`NULL` when it should have been `NULL`, and the kernel panicked.

Even with this assert, extremely mysterious behaviour could have arisen in other use cases, such as during stress testing: if the awakened thread had gone to sleep on another condition variable, its `cond_queue_next` pointer could point to a thread asleep on a different condition variable, in which case a broadcast on the new condition variable would cause a spurious wakeup of the second thread. Worse yet, if the second thread had also been awoken, and thence exited, a broadcast would result in scheduler data structure corruption.

This bug is exposed with the `double_wait` test case. The complexity of this bug is moderate, since it requires both waiting threads to block before the child process exits. The fix is simple; in two places in the condition variable code a pointer needed to be set to `NULL`.

We note that the author never knew of its existence between submitting the kernel as a student three years ago and uncovering it before the user study three weeks ago, and it was not spotted by the TA who graded it.[2]

[2]We would also note that this bug was never uncovered by any stress test, but the `double_wait` test case was not distributed as part of the 15-410 test suite, and it was necessary to generate the right sequence of system calls.

POBBLES thread_fork/wait

This bug was presented in the lecture, and exhibited as a decision tree in Figure 4.1 and as a trace in Figure 5.5 and as source code in Figure 6.1. When a new thread is forked and made runnable, and the forking thread subsequently dereferences its TID to use as a return value, it's possible for the dereference to be a use-after-free. This could result in a garbage value being returned.

This bug is exposed with the `double_thread_fork` test case. The complexity is relatively simple,[3] and the fix is only two lines long; the dereference should simply occur before making the child thread runnable.

Nevertheless, this bug was present in the author's student kernel and was not noticed by the TA who graded it.

LudicrOS vanish/vanish

This bug was found by the author when grading this kernel. It took roughly an hour of manual inspection and hard thinking to ascertain the bug's nature, from the time when the author first suspected a bug might be present.

The parent pointer problem is the same as in the POBBLES vanish/vanish bugs, except the locking pattern is slightly different. When the parent reparents children, it drops its own mutex while it changes the parent pointer using the child's parent pointer mutex, which happens before the parent moves the child to init's child-list.

Many problems can arise from this error. The one Landslide discovered was as follows: During this gap, the child can take its own parent pointer mutex (after the pointer was changed to init), and attempt to move itself from init's "live child-list" to the "dead child-list" before the parent moved the child to init's child-list at all. This results in the child instead moving the *shell process* off of init's "live child-list". Init gets signalled, reaps the reparented child, does `wait` again (expecting the shell to be alive), which fails. This results in init infinitely looping around failing calls to `wait`. To catch this bug immediately, rather than relying on infinite loop detection, we added an assert statement in the `wait` failure case that the current thread was not init (because init should always have children to reap or wait for), which Landslide then caused to trip.

This was the most complicated bug we studied. The fix would be nontrivial, like in POBBLES vanish/vanish.

LudicrOS yield/vanish

This bug was also found during grading, and took substantially less time to verify by hand than the previous bug. When a thread does `yield` to a specific other TID, the other thread's

[3]Many userspace thread libraries forcibly serialise invocations of `thread_fork` and `vanish` with their own synchronisation barriers, so the kernel bug may never be noticed when using such libraries (this caused the author some trouble).

Kernel and test case	Decision points used			
	default	lock	unlock	both
POBBLES vanish/vanish (a)	[*31.8 (0.6)*]	**57.1** *(1.7)*	∞	∞
POBBLES vanish/vanish (b)	[*32.0 (0.4)*]	**51.5** *(2.1)*	[*8057.9 (336.9)*]	∞
POBBLES wait/wait	**23.3** *(0.7)*	**27.9** *(0.8)*	**27.9** *(2.1)*	**41.6** *(1.4)*
POBBLES thread_fork/vanish	**22.0** *(0.6)*	**37.4** *(1.1)*	**27.6** *(0.5)*	**72.0** *(2.6)*
LudicrOS vanish/vanish	[*13.2 (0.2)*]	**13.7** *(0.7)*	**34.6** *(1.1)*	**17.1** *(0.3)*
LudicrOS yield/vanish	[*12.3 (0.3)*]	**11.4** *(0.4)*	[*27.4 (0.8)*]	**11.7** *(0.4)*

Table 7.2: Comparison of time taken (in seconds) to find bugs using Landslide with various decision sets: the default set, consisting only of voluntary reschedules (Section 5.2.4); and using custom decision points in addition to the default set: calls to `mutex_lock`, calls to `mutex_unlock`, and both. All numbers represent the average from 5 trials, with the standard deviations given in parentheses.
Key: **seconds** *(stddev)* indicates bug found; [*seconds (stddev)*] indicates whole tree explored with no bug. "∞" indicates that Landslide's search did not finish (after 8 hours).

thread control block (TCB) is not protected after being returned from the look-up function. If that thread exits during that time, a use-after-free (and worse, a completely invalid schedule to a nonexistent thread) results.

This bug is moderately complicated, requiring a preemption during that specific window of the first thread's execution during which the second thread must exit entirely and have its TCB freed. The fix is moderately complicated as well; since simply reordering lines of code won't offer the necessary protection.

7.2.2 Performance

Table 7.2 shows the time it takes to find each of these bugs using Landslide, configured with several different sets of decision points. Figure 7.2 compares the total exploration time among each different decision set for the trials that did find bugs.

The experimental set-up is as follows:

- All Landslide trial times include the Simics start-up and kernel boot-up time (time between issuing the command and the test case beginning to run), roughly 15 seconds for POBBLES and 10 seconds for LudicrOS.

- All Landslide trials were run on the Gates-Hillman cluster machines (2.6 GHz Xeon; four cores, though only one was used; 8 GB RAM).

- All Landslide trials were run with "backwards exploration" enabled (Section 6.3.2).

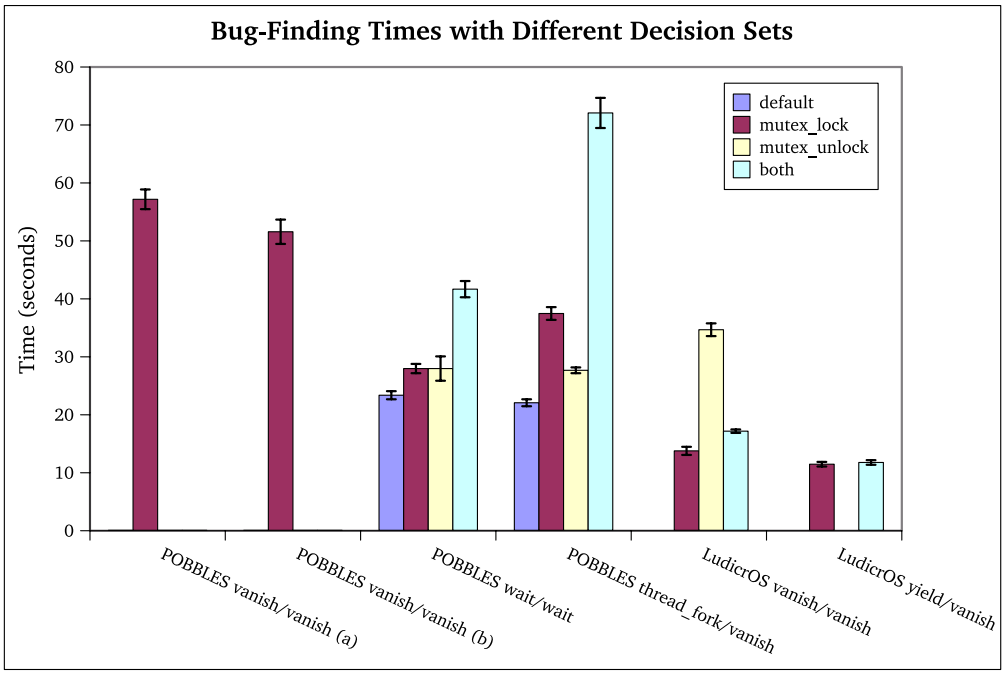


Figure 7.2: Bar graph visualisation of the exploration time in trials that did find bugs. The decision trees that did not expose bugs and the ones that timed out are not shown.

- All trials were also run with Landslide configured to pay attention to only the relevant system calls (using `within_function`; Section 6.3.1).

  We believe it is reasonable to test for these bugs in this way (using the minimal set of system calls to be paid attention to as necessary to find the bug) because it follows the recommended workflow of using Landslide, which is to start with what the user judges to be the "smallest relevant set" of decision points. The configuration using `within_function` was as follows.

  - POBBLES vanish/vanish(a): `within_function vanish`
  - POBBLES vanish/vanish(b): `within_function vanish`
  - POBBLES wait/wait: `within_function wait`
  - POBBLES thread_fork/vanish: `within_function thread_fork` and `within_function vanish`
  - LudicrOS vanish/vanish: `within_function vanish`
  - LudicrOS yield/vanish: `within_function yield` and `within_function vanish`

Table 7.3 presents more detailed information about the decision trees that Landslide explored when finding these bugs. For each set of decision points on each bug, we give the total number of decision points in the tree, the total number of backtracks (i.e. branches explored before the bug was found), and the average branch depth (i.e. number of decision points in each branch).

Kernel and test case	Property of tree	Decision points used			
		default	lock	unlock	both
POBBLES vanish/vanish (a)	Decision points	[56]	1296	N/A	N/A
	Total backtracks	[16]	376	N/A	N/A
	Average branch depth	[5]	19	N/A	N/A
POBBLES vanish/vanish (b)	Decision points	[56]	1295	[382071]	N/A
	Total backtracks	[16]	376	[112706]	N/A
	Average branch depth	[5]	17	[16]	N/A
POBBLES wait/wait	Decision points	23	102	74	378
	Total backtracks	4	17	12	56
	Average branch depth	6	10	9	14
POBBLES thread_fork/vanish	Decision points	24	394	273	2269
	Total backtracks	5	70	56	410
	Average branch depth	5	16	14	23
LudicrOS vanish/vanish	Decision points	[10]	16	141	42
	Total backtracks	[2]	3	48	10
	Average branch depth	[2]	7	9	14
LudicrOS yield/vanish	Decision points	[8]	5	[149]	7
	Total backtracks	[1]	0	[43]	0
	Average branch depth	[2]	0	[9]	0

Table 7.3: Information about the decision trees explored when finding bugs. As in the previous table, each test case was run with the four different sets of decision points. "[X]" means the tree was completely explored because Landslide did not find a bug in that configuration. "X" reflects the portion of the tree that was explored before a bug was found.

7.3 Discussion

7.3.1 Invariants

While evaluating Landslide on these bugs, we determined two invariants that must hold for multiple explorations on the same test case.

1. **Ordering invariant.** For a given set of decision points, exploring the tree in multiple different orders ("forwards"/"backwards") should produce the same result in terms of whether a bug was found or not. The bugs found may be different, and the number of branches explored may be different, but it should never be that one ordering finds a bug while another ordering of the same tree finds no bug.

2. **Superset invariant.** For a given set of decision points, if an exploration of the resulting tree finds a bug, using a superset of that set of decision points should also find a bug. This is because the first tree will be a sub-tree of the second, as shown by never preempting at a decision point that appears in the second set but not the first.

In short, even though Landslide may give false negatives from using imperfect sets of decision points, the exploration itself must be sound (i.e. not missing any bugs that exist in the resulting tree).

When running LudicrOS yield/vanish with decision points on `mutex_unlock` but not on `mutex_lock`, we found that the ordering invariant failed: "backwards" exploration found no bug, while "forwards" exploration did. We attribute this to a bug in Landslide itself, and present the results for the backwards exploration as usual, in which Landslide found no bug.

7.3.2 Recommended Testing Strategies

We make several observations about the experimental results from Section 7.2.2.

1. **Fewer is faster.** While it is theoretically possible that a tree built of finer-grained interleavings might encounter a bug after fewer overall backtracks, we found that this did not happen in practice.[4] In general, for two sets of decision points that both find the same bug, the one that results in shorter *branches* will result in fewer *backtracks* needed to expose the bug, and hence less overall time.

2. **Different decision points are differently likely to expose different bugs.** We see that even though the "lock" and "unlock" trees tended to be about the same size, they were each sometimes better than the other at finding bugs. The "lock" tree did better on the vanish/vanish bugs and the yield/vanish bug, while the "unlock" tree did better on the wait/wait bug and the thread_fork/vanish bug.[5]

3. **Finding a bug is fast, if it exists.** As especially exhibited in the POBBLES vanish/vanish (b) case, if two sets of decision points generate trees of approximately equal size, but one tree contains a bug and the other doesn't, then searching the bugful tree will likely terminate much more quickly. Of course, it is always possible that a bug may exist only in the very last branch of a tree, but we found that in general bugs show up early during exploration.

In light of these, we recommend several principles to govern an overall strategy to automatically iterate through different sets of decision points in search of a bug.[6]

[4]The one suspicious case is the LudicrOS vanish/vanish bug. Exploring the "both" tree found the bug faster than exploring the "unlock" tree, despite the latter's decision points being a subset of the former's. However, we see that exploring the "lock" tree found the bug faster than either, so overall the "both" tree did not outperform the fastest of the smaller trees.

[5]The latter two bugs needed no more than the default set to uncover at all, but we claim this still demonstrates the principle in general.

[6]We now say "Landslide" here to refer to a hypothetical test framework to embody these strategies, although of course it would not have to be named that.

1. **Iterate exploring, starting with smaller decision sets.** To properly test for a bug in a particular test case, Landslide should try as many different decision sets as possible. The first exploration should always be just the default decision set, because that tends to complete quickly, and can help identify new decision points (Section 8.3.1). When Landslide has multiple decision sets as candidates for the next exploration, it should prefer to explore ones that result in shorter average branch depth. In this way, Landslide will tend to find bugs with the minimal decision set needed to expose them, in accordance with observation 1 above.

2. **Run multiple explorations at once.** As per observation 2, if Landslide has two decision sets that result in roughly equal average branch depth, it cannot know in advance whether either one will find a bug and/or finish significantly faster than the other. As such, it should try to run both explorations in parallel, wait for either one to finish, and continue iterating as appropriate even if the other one has not finished.

3. **De-prioritise longer-lasting test configurations.** With finite resources for parallelisation, Landslide should attempt to load-balance whatever searches are running in parallel according to each one's likelihood of finding a bug. In light of observation 3, if a particular search is running abnormally long for its average branch depth (the POBBLES vanish/vanish (b) bug with the "unlock" tree is a prime example), Landslide could judge that it is less likely to end soon with a positive result, and prioritise searches with other decision sets.

Chapter 8

Future Work

As Landslide is an experimental foray into the world of systematic exploration in kernel-space, and a detailed study of systematic exploration in general, it has opened up many avenues for potential future improvements.

8.1 Interface Improvements

In Section 7.1.2, we describe many ways in which Landslide's user interface is lacking. These improvements would not necessarily be research developments, but would enhance the user experience in ways necessary for continuing work in the context of 15-410.[1]

- **Replay.** Landslide currently does not support ability to replay a particular decision sequence. If the user finds a bug, and wants to re-execute the interleaving that led to it, they have no choice but to re-explore the tree.
- **Debug prompt support.** Although Simics can pause execution and give the user a debug prompt, Landslide currently does not support being interrupted during exploration. This is because the current implementation uses a wrapper script around the Simics prompt which Landslide communicates with to perform backtracking. Though it would be difficult to redesign, it would be good to work around, to enable the user to interrupt, use the Simics debug prompt arbitrarily, and resume Landslide when ready.
- **Decision trace formatting.** Landslide could print its decision traces in a better format. The current format is basically a "raw data dump". Raw text is probably the wrong interface; a graphical/tabular format with distinct columns for the execution of each different thread would be easier to understand.

[1] Landslide is also missing functionality to automatically identify decision points when newly-forked threads become runnable but not immediately switched to. This was not necessary to find the `double_thread_fork` bug in POBBLES (Section 7.2) only because POBBLES automatically switched to the child thread immediately.

- **Feedback for incorrect instrumentation.** Landslide could attempt to detect when something has gone wrong that is not its fault (such as incorrect instrumentation, or the kernel violating some of its requirements), and print a useful error message rather than behaving mysteriously.

8.2 Education

8.2.1 Landslide as a Teaching Tool

Figure 8.1: Former members of Operating Systems course staff lament the difficulty involved in teaching students concurrency debugging skills.[1]

15-410 currently teaches students concurrency debugging skills "the hard way": by immersing them in environments where races will arise, and letting them find debugging tactics on their own to use in conjunction with conventional stress testing.

We believe that a tool such as Landslide could change the way students learn such skills for the better, beyond simply increasing their likelihood of finding bugs during testing. The

[1] Joshua Wise, Ben Blum, and Michael Sullivan at Vail, Colorado. Included with permission.

decision tree is a much more structured way of expressing a test case's concurrent behaviour than the way 15-410 currently teaches, which is "think hard until you figure out the buggy interleaving." Having students use Landslide, even to a small extent, might encourage them to think in this more structured way.[2]

8.2.2 Landslide as a Grading Tool

The 15-410 grading infrastructure currently uses a program called Fritz, which is a stress testing wrapper around a suite of test programs. Some of these tests are Landslide-friendly (Section 6.4.1), and some are themselves stress tests. The 15-410 course staff also hand-grade every kernel, in part because Fritz is known for only catching race conditions at random and infrequently.

While far from being able to replace talented humans for grading, we believe that, after some improvement, Landslide could augment or replace Fritz as a tool for automatically finding several common patterns of concurrency errors that students have (especially races involving exiting threads, which we have shown Landslide is effective at finding).

8.2.3 Making Pebbles Landslide-Friendly

If Landslide becomes integrated into an undergraduate course curriculum, it will be important to reduce the amount of effort students must go through to instrument their kernels. This could be achieved in part by changing the project definition in ways that make Landslide better able to automatically instrument kernels. For Pebbles, we recommend the following changes.

1. **Timer ticks control "runnable" threads.** As discussed in Section 5.1.2. Additionally, to prevent the users from having to instrument non-descheduling mutexes, the project could mandate that all concurrency primitives actually remove blocked threads from the runqueue.

2. **No idling when progress can be made.** Also as discussed in Section 5.1.2. With the current specification, inappropriate idling is merely a performance bug, though it is incompatible with Landslide.

3. **Mandated function names for scheduling events.** As part of the instrumentation process, students tell Landslide where in their schedulers notable events happen. Landslide could automatically detect some of these if the project mandated the names of certain functions, such as the timer handler, context switcher, and runqueue manipulation routines.

[2] In 15-410, before the kernel project, students also implement a user-space thread library. Just as Landslide could help the students during the kernel project, so too could a user-space systematic exploration tool help during the thread library project.

8.2.4 Virtual Memory Bug-Finding

In this work, we focused especially on races related to the thread lifecycle routines. We believe Landslide would be immediately applicable to certain varieties of virtual memory bugs, but other types might not be easily exposed with Landslide's current model.

For example, one of the groups in the user study said they were looking for a race that only arose in out-of-memory conditions. A test case that could expose that race, by exhausting the kernel's memory resources, would necessarily have a very large state space that would be impractical to explore systematically.

We suggested that the group configure their kernel to fail memory allocations every so often, so Landslide would be better able to find the race while still using a small test case. To make Landslide more applicable to such bugs in general, we might further investigate this debugging strategy in a more structured way.

8.3 New Techniques

Related research in dynamic verification has introduced testing techniques orthogonal to systematic exploration, which could augment its effectiveness if combined in one tool. We discuss the potential for using other techniques in Landslide or in any other systematic testing framework.

8.3.1 Data Race Detection

Landslide currently never alters its set of decision points once it begins executing. The user must configure the set of decision points in advance, and Landslide follows them to the letter when exploring the tree. This is useful for investigating the size of trees generated by certain sets of decision points, but future work should do better than leaving it up to the user to stumble across a set of decision points that exposes a bug.

Most notably among the information Landslide has at its disposal is the collection of conflicting shared memory accesses among transitions. Theoretically, to identify a decision point immediately before and after each such access would generate an execution tree with perfect granularity (i.e., exactly one shared memory access per transition), and thence find every possible race condition, which would be the opposite extreme to the current setup.

Landslide could make use of data race detection techniques [EMBO10], however, to strike a middle ground in which shared memory accesses it chooses.

1. It should be aware of the types and interfaces associated with the kernel's synchronisation primitives (mutexes, condvars, semaphores, etc), and be able to treat operations on those as "guaranteed to work" (similar to Section 5.4.2).[3]

[3]If such operations were instead themselves incorrect, it would mean that data race detection might fail to identify certain racy accesses. The technique would then still identify some racy accesses, but might miss others.

2. Next, it should recognise acquire and release (or sleep and wake) operations on synchronisation primitives, and use that information to track lockset-type information.

3. Ultimately, it should use the lockset information in conjunction with the happens-before relation to identify which memory accesses are "raciest", and add decision points around those.

8.3.2 Parallelism

Landslide's tree exploration is implemented sequentially. However, because DPOR's approach of tagging which sibling branches should be explored next generally follows a workqueue-based depth-first-search structure, it should not be too difficult to parallelise. Prior work exists for this technique [YCGK07], so it would not be a research contribution, but would substantially improve Landslide's effectiveness regardless.

8.3.3 Symbolic Execution

In Section 2.3, we mentioned the technique of symbolic execution for dynamic verification. We believe that symbolic execution could be combined with systematic exploration to help find races in certain obscure code paths that systematic exploration by itself might not even execute.

One interesting phenomenon that would occur from combining symbolic execution and systematic exploration would be a new notion of decision points. Symbolic execution involves its own state space exploration, in which control flow branch points correspond to decision points; hence, to combine it with systematic exploration would create a hybrid state space in which a "decision point" could be a place where either different threads could preempt each other or multiple control flow paths could be taken [Eng12].

8.3.4 Trace Minimisation

Also in Section 2.3, we mentioned related research that attempts to *minimise* the set of conditions that are known to be associated with a bug [SKM+11]. In the context of Landslide, this corresponds to how easy the decision trace is to understand. Even if Landslide finds a bug and prints the list of thread switches that were made, it may still be very difficult for the user to understand if the list includes many unnecessary switches that were completely unrelated to the bug.

Landslide could continue to be of use even after it has found a bug, by attempting to find a "minimal decision trace". It could search different subsets of the known-buggy trace in an attempt to find shorter traces which result in the same bug. In this way, it would be able to print decision traces that automatically give the programmer much more insight into the true nature of each race condition.

While studying various bugs found by Landslide, we found that in general, when there were frivolous thread switches in the decision trace, users most benefitted from starting to read the trace at the end; i.e., considering the most recent transition made by each thread. Hence, when building a search algorithm for trace minimisation, it would be good to more heavily prioritise decisions that occurred later in a buggy branch.

8.4 Linux

The Linux kernel is a logical next kernel architecture to target with Landslide.

One advantage of targetting Linux is that a testing framework will be able to make assumptions about the scheduler design, in ways that Landslide was not able to (Section 3.3), because the kernel instrumentation will not need to be repeatedly re-implemented (as it did for each 15-410 student kernel that we tested).

However, Linux is a much more complicated kernel architecture than Pebbles, and we would face several challenges.

8.4.1 Multi-Processor Support

Landslide's way of modelling concurrency as a tree of thread interleavings directly expresses the way concurrency arises on uniprocessor systems. On multiprocessor systems such as Linux, however, there is "true concurrency": different threads may be running at the same time on different processors, rather than just interleaved. Hence, race conditions that were previously impossible in a uniprocessor environment may be commonplace in SMP.

In order to support multiprocessor kernels, Landslide would need to refine its representation of the decision tree to express the potential for threads interleaving either on the same processor or on different processors. As a starting point, we note that on SMP, at each decision point, in addition to choosing any of N threads to run, we must choose from among M processors for it to run on.

8.4.2 Performance

Linux is much bigger than a Pebbles kernel (in terms of amount of code that runs during startup, during scheduling, etc), and hence would be much more expensive to test in a simulated environment. We suspect that in order to practically test Linux, we would need to implement systematic exploration in a virtualised environment, which we discuss more in Section 8.5.

8.4.3 Complicated Synchronisation Patterns

Many components of Linux contain ad-hoc synchronisation, often done for performance reasons, which would be difficult to reason about. One common pattern is attempting some operation, checking later if it got interfered with, and rewinding and trying again if so. A straightforward approach to building a decision tree representing this might result in an infinitely deep branch: if the thread that gets interfered with keeps getting selected for scheduling, it would never make any progress. The testing tool would need to recognise these cases, and understand the invisible dependency on "somebody else" running first.

8.4.4 Device Drivers

There is much more device driver code in Linux than there is for its core components, and the device driver code also does not undergo as rigorous code review. Because of this, device drivers are much more prone to concurrency errors than the core components, and hence a systematic exploration framework for Linux should focus on testing them.

However, the concurrency model of device drivers is much more complicated than the simple timer-based scheduling model we used in this work. In addition to timer-driven preemptions, code may also be interleaved as caused by device interrupts and input. Like support for SMP, being able to systematically test for device driver races will also require a more sophisticated representation of the decision tree.

In addition to controlling timer interrupts, a systematic testing tool would also need to control device interrupts and device input, both of which are also nondeterministic inputs to the kernel. Furthermore, it would need to recognise common components of device drivers to properly cause relevant scheduling patterns, such as the "top half" and "bottom half" of interrupt handlers and dedicated device driver threads.

These components would all be candidates for being scheduled at a decision point, although it more complicated still than simply choosing from among several threads to run. For example, the bottom half of an interrupt handler would depend on a top half having already executed, and interrupt handlers would run on the same stack as other thread rather than on their own stack.

Because many driver bugs result from interactions with error-handling paths, it will also help to understand the difference between normal device input and faulty input in order to maximise code path coverage. Symbolic execution (Section 8.3.3) may help address this.

8.5 Virtualisation

Compared to executing the code being tested on real hardware (which tools such as [SBG10] are able to do for user-space programs), simulation in Simics is slow, and even slower with Landslide hooked up to it.

Major benefit could be achieved by redesigning Landslide to work in a virtualised environment instead of a simulated one. This would pose several challenges, because Landslide would no longer be running in "omniscient mode", having immediate access to every single instruction and memory access executed. It would, however, still be able to modify the kernel's memory and address space, which would be important to harness for increased control over the kernel's execution.

We speculate on some challenges and potential solutions that virtualised execution might have to offer. We assume that our users will be talented kernel developers rather than undergraduate students, and hence allow for more complicated interface requirements.

8.5.1 Interposition

Event Tracking

The event-based nature of the `tell_landslide` annotations can still be harnessed in a virtualised environment. They would need to be converted into hypercalls, which would briefly transfer control to Landslide at key execution points to enable it to update its state machine and potentially influence the kernel's execution.

We would add additional `tells_landslide` for the kernel components which are presently automatically instrumented, such as the dynamic memory allocator and the `panic` routines, and also for certain components that are currently instrumented with `config.landslide`, such as identifying when the context switcher returns or when the timer interrupt handler is invoked.

Rather than using the schedule-in-flight algorithm, Landslide could simplify its job by asking the kernel to run a particular thread directly, as described in Section 5.2.2.

Memory Tracking

More challenging than tracking important kernel events will be tracking all accesses to shared memory. We could exploit the virtualised virtual memory system, and write-protect (possibly also read-protect, i.e., unmap) all pages that contain shared data, such as the global data section and the dynamic allocation heap. All accesses would then page-fault, transferring control to Landslide.

8.5.2 Backtracking

Simics's convenient bookmarking mechanism will be no more in virtual-machine-land. One possibility for backtracking would be to record-and-replay the events of each branch (up to the point at which Landslide chooses to have decided differently), restarting the system's execution from the beginning each time. Another option might be to snapshot the state of the kernel (and associated devices) at each decision point, and use those to resume without

re-executing everything from the beginning (kernel boot-up and initialisation is much more expensive than individual system calls, after all).

8.5.3 Control over Non-Determinism

Finally, Landslide will need to find new ways to control non-deterministic events in the system. Since timer ticks in virtualised kernels are mediated by the hypervisor, Landslide may need to intercept the timer, and perhaps control the guest kernel's context switching with a custom "Landslide-driven" context switcher instead.

Likewise, Landslide will need to control other non-deterministic inputs to the kernel, including device interrupts/input and even the system clock.

8.6 Long-Running Test Shaping

Another area to investigate is that of systems which could be tested continuously. Instead of the goal of finishing exploring a test case without finding a bug, a long-running testing approach would focus on "shaping" the test configuration to more and more refined configurations in hopes of finding a bug eventually.

In Section 7.3.2, we recommend an iteration strategy to effectively explore trees with multiple different sets of decision points, while not knowing ahead of time which, if any, may contain a bug. This strategy would give rise to a test framework which could explore multiple different types and granularities of interleavings at once, heuristically judge which are more likely to uncover bugs, and prioritise resource allocation and search direction accordingly.

There is also the question of when during the testing process new decision points should be introduced. One possibility would be to begin exploration with a small set of decision points, explore the resulting tree, analyse the tree post-hoc to identify additional decision points, and iterate exploring with the new set. Another possibility would be to identify decision points along the way, analysing each branch of the tree after executing it, to generate the decision points for that very branch, and thence use them to find which branch to explore next. It remains to be seen which approach would be more effective.

One possible application of such a strategy would be in searching for bugs given only a stack trace and/or kernel log from a crash report. The components of the kernel mentioned in the trace or log could be used as the initial focus for the test, and the framework could gradually expand its scope to search for an interleaving granularity which would reproduce the bug.

Bug	Direction	Time	Backtracks	Trace Length
POBBLES vanish/vanish (a)	Forwards	124.8 *(4.9)*	1264	7
	Backwards	57.1 *(1.7)*	376	11
POBBLES vanish/vanish (b)	Forwards	106.3 *(5.7)*	1030	7
	Backwards	51.5 *(2.1)*	376	10
POBBLES wait/wait	Forwards	62.6 *(3.4)*	132	8
	Backwards	27.9 *(0.8)*	17	13
POBBLES thread_fork/vanish	Forwards	21.4 *(0.7)*	0	6
	Backwards	37.4 *(1.1)*	70	6
LudicrOS vanish/vanish	Forwards	14.7 *(0.9)*	4	3
	Backwards	13.7 *(0.7)*	3	7
LudicrOS yield/vanish	Forwards	13 *(0.9)*	3	3
	Backwards	11.4 *(0.4)*	0	5

Table 8.1: Comparison of forwards and backwards explorations for the six case study bugs (using decision points on `mutex_lock` but not `mutex_unlock`).

8.7 Theoretical Oddities

During our in-depth case studies, we noticed two interesting properties about certain decision trees. We believe these warrant further theoretical study, to better understand the nature of these decision trees. This may in turn give rise to heuristics for making systematic testing more effective.

8.7.1 "Backwards" Exploration

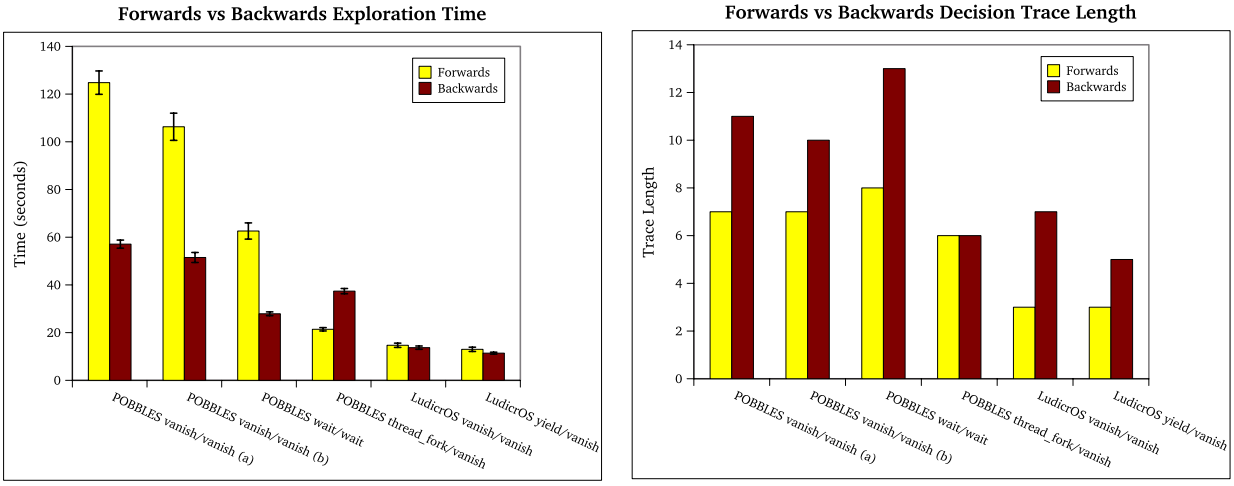


Figure 8.2: In general, backwards exploration finds bugs faster, but forwards exploration produces shorter (more understandable) decision traces. (Lower is better in both graphs.)

We configured Landslide to be able to explore trees in two different orderings, and we found that the two orderings behaved substantially differently in terms of time taken to find bugs and length of decision traces produced.

- **Forwards exploration.** When exploring the tree "forwards", at each decision point the explorer would choose the currently-running thread if it hadn't already been explored. This resulted in preference for branches of the tree with fewer forced preemptions; the first branch to be explored would have no preemptions at all, and subsequent branches would have generally (but not monotonically) increasing numbers of preemptions.

  Compared to backwards exploration, we found that forwards exploration tended to produce much shorter decision traces for the same bugs. These shorter traces would have fewer thread switches that were irrelevant to the bug itself, and hence be much easier to understand.

- **Backwards exploration.** When exploring "backwards", at each decision point the explorer would prefer to choose a thread that required a forced preemption. This resulted in preference for branches with more forced preemptions, and the first branch to be explored would have a preemption at every decision point.[4]

  Compared to forwards exploration, we found that backwards exploration tended to find bugs much more quickly, with fewer backtracks needed before the buggy branch was chosen.

Table 8.1 and Figure 8.2 compare the tree explorations and resulting decision traces. The one exception to backwards being faster is the POBBLES thread_fork/vanish bug, which Landslide found immediately during forwards exploration, because the race actually required no forced preemptions to expose.

Compared to Iterative Context Bounding [MQB+08], which is based on the insight that bugs need fewer forced preemptions to expose, we believe that our insight is orthogonal. The ICB insight results in smaller decision trees overall needing to be explored before bugs appear, while our insight determines exploration ordering once the tree is constructed.

We are not sure if combining these two insights is possible. In future work we might compare them and investigate this possibility, to determine a heuristic for what exploration orderings are most likely to expose bugs with the fewest needed preemptions.

8.7.2 Exploration Tree Structure

One member of Group 1 from the user study fully explored decision trees from `double_wait` and `vanish_vanish` (Section 7.1.3), and found an oddity in the exploration time for a particular decision set.

[4]This property of backwards exploration is what resulted in the LudicrOS yield/vanish bug being found without any backtracks.

When using decision points on just `mutex_lock` or `mutex_unlock`, the explorations for each test case took similar time. However, when using decision points on both mutex functions, `double_wait` took substantially longer, despite the tree structure suggesting they should have been comparable, with `double_wait` even being perhaps slightly faster. Table 8.2 shows a comparison of the tree anatomy for these two test cases.

	`double_wait`	`vanish_vanish`
Exploration time	**30 minutes**	**5 minutes**
Backtracks	**1,401**	**1,511**
Decision points	10,355	5,690
Average branch depth	18.34	16.00
Average instructions/branch	**1,975,033**	**1,984,249**
Total instructions	206,757,601	24,907,516
Backtrack distance minimum	**6**	**2**
Backtrack distance maximum	17	16
Backtrack distance average	**7.38**	**3.75**

Table 8.2: Comparison of expensive `double_wait` test and cheap `vanish_vanish` test. The number of branches and depth of each branch are similar, but `double_wait`'s backtracks were significantly longer, causing more total instructions to be executed.

Despite the two trees having nearly equal numbers of branches and instructions executed per branch, `double_wait` took significantly more time. This is because Landslide's backtracks during `double_wait` were significantly *longer* than the backtracks in `vanish_vanish` on average; i.e., the backtracks were targetting an earlier point during the test's execution. Hence, Landslide had to execute more total instructions overall to explore the longer branches in `double_wait`.[5] We show an exaggerated visualisation of this tree structure in Figure 8.3.

This suggests that the "interesting" parts of `double_wait` were earlier in the test case, whereas in `vanish_vanish` they are more towards the end. We conjecture that, with theoretical study of these structural properties, we could develop search heuristics for focusing on the "interesting" parts of the trees, and avoid repeating the irrelevant instructions at the end of each branch.

[5]Of note, if backtracking had been implemented by restarting the test each time; i.e., executing the full depth of every branch from the root of the tree, the repeated work would have caused the execution times to be equal. The fact that Simics's backtracking avoids the cost of execution that happened before the backtrack target is to credit for this discrepancy.

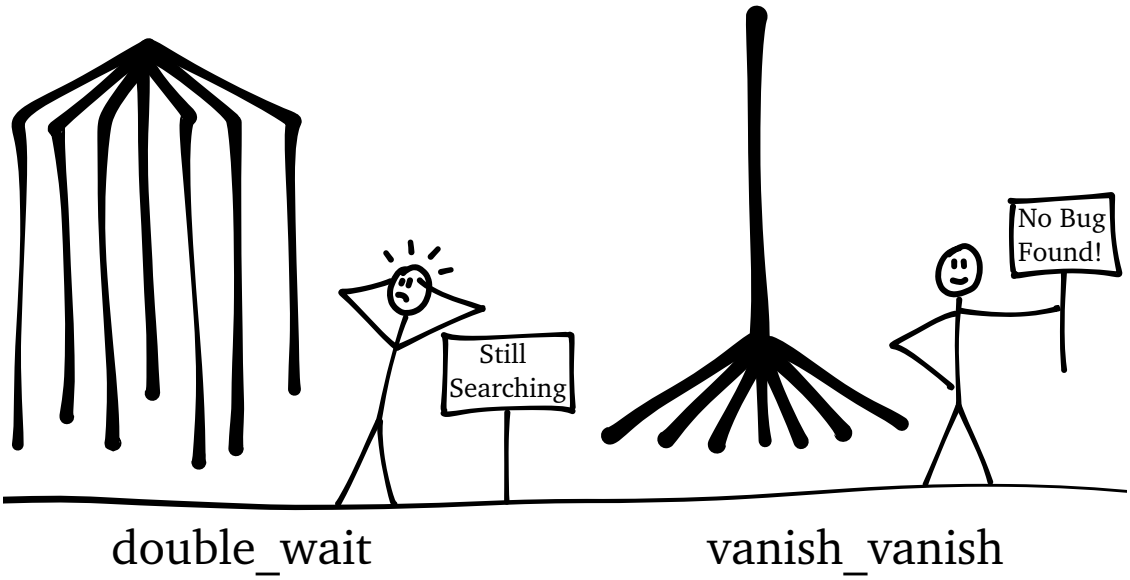


Figure 8.3: Though the execution trees in `double_wait` and in `vanish_vanish` had the same number of branches, with each approximately as many instructions deep, the backtracks in `double_wait` were much longer, resulting in more *total* instructions executed during the exploration.

Chapter 9

Conclusion

Systematic exploration is a powerful technique for exposing race conditions in concurrent systems. In previous research, systematic exploration has proven useful in user-level code and distributed systems, yet a gap stands between the technique and the complex world of kernel-level concurrency debugging. In this thesis, we present techniques for bridging this gap.

We have built Landslide, a tool for performing systematic exploration in kernel space with a focus on Pebbles kernels written for 15-410. We showed that Landslide's techniques make systematic exploration in kernel space both possible and efficient, and that Landslide can help students find bugs in their own kernels.

With Landslide, we see testing a kernel as a process of manipulating test parameters in two ways: first, in the choice of test case, and second, in the user's configuration of Landslide to express which parts of the kernel are "interesting" and which are irrelevant. Finding and understanding race conditions exposed by a given test becomes a joint effort between the user and Landslide, combining the user's knowledge about the kernel's design and Landslide's ability to explore many interleavings efficiently.

Our work on Landslide has indicated many possible avenues for continued work on systematic exploration in kernel space. We see potential for Landslide to be a useful debugging tool both for students of 15-410 and for developers of general purpose operating systems. Application to mainstream kernels also opens up the possibility to apply systematic testing techniques to device driver code, known for being prone to concurrency errors. Finally, the current emphasis on user-guided steering of test parameters suggests long-running testing approaches which could automate a user's intuition for how to find meaningful results quickly.

We hope that the ideas presented here serve as stepping stones for future development of more sophisticated systematic debugging techniques for kernel-level race conditions.

> *And lo, the Author didſt preſent Landſlide to yon reſearch Community, and to the ſtudents of XV-CDX as well. Verily, the Kernel developers young and olde found Bugges of Concurrenſy in their code. The ſoftware thenſeforth worked as Intended, craſhing nevermore, and there was much rejoicing.*

Bibliography

[ACHB11] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic Exploit Generation. In *Network and Distributed System Security Symposium*, February 2011.

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[CWG+11] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 337–351, New York, NY, USA, 2011. ACM.

[Eck11] David Eckhardt. Personal communication, 2011.

[EMBO10] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[Eng12] Dawson Engler. Personal communication, 2012.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[God97] Patrice Godefroid. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 476–479, London, UK, UK, 1997. Springer-Verlag.

[GWZ+11] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 265–278, New York, NY, USA, 2011. ACM.

[KAJV07] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: finding liveness bugs in systems code. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, pages 18–18, Berkeley, CA, USA, 2007. USENIX Association.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[KRS09] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating hardware device failures in software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 59–72, New York, NY, USA, 2009. ACM.

[LCB11] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM.

[LVT+11] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 353–367, New York, NY, USA, 2011. ACM.

[MCE+02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, February 2002.

[MQB+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[OAA09] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 97–108, New York, NY, USA, 2009. ACM.

[SBG10] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: systematic evaluation of distributed systems. In *Proceedings of the 5th international conference on Systems software verification*, SSV'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[SBGH11] Jiri Simsa, Randy Bryant, Garth Gibson, and Jason Hickey. Efficient Exploratory Testing of Concurrent Systems. *PDL-CMU Technical Report*, 113, November 2011. `http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-11-113.pdf`.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[SKM+11] Eunsoo Seo, Mohammad Maifi Hasan Khan, Prasant Mohapatra, Jiawei Han, and Tarek Abdelzaher. Exposing Complex Bug-Triggering Conditions in Distributed Systems via Graph Mining. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 186–195, Washington, DC, USA, 2011. IEEE Computer Society.

[YCGK07] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *Proceedings of the 14th international SPIN conference on Model checking software*, pages 58–75, Berlin, Heidelberg, 2007. Springer-Verlag.

[YCGK08] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. In *Proceedings of the 15th international workshop on Model Checking Software*, SPIN '08, pages 288–305, Berlin, Heidelberg, 2008. Springer-Verlag.

[YCW+09] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[YRC05] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 221–234, New York, NY, USA, 2005. ACM.

[YSaR12] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. SimTester: a controllable and observable testing framework for embedded systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 51–62, New York, NY, USA, 2012. ACM.

[YST+06] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically Generating Malicious Disks using Symbolic Execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 243–257, Washington, DC, USA, 2006. IEEE Computer Society.

Appendix A

Code for Provided Test Cases

The test cases that use the `thread_fork` system call are written in assembly, because thread creation can only safely be done from C using a user-space thread library. However, using a user-space thread library for thread creation might forcibly serialise certain operations that would expose kernel bugs only if concurrent, and also necessitates many more system calls, which dramatically increases the size of the decision tree.

A.1 vanish_vanish.c

```
#include <syscall.h>

void main()
{
    fork();
    vanish(); /* parent and child process exit simultaneously */
}
```

A.2 double_wait.S

```
#include <syscall_int.h>

.global main
main:
    int $FORK_INT
    cmp $0x0,%eax
    jne parent
    int $VANISH_INT
parent:
    int $THREAD_FORK_INT
```

```
    mov $0x0,%esi
    int $WAIT_INT        # two parent threads do wait(NULL);
    int $VANISH_INT
```

A.3 yield_vanish.S

```
#include <syscall_int.h>

.global main
main:
    int $THREAD_FORK_INT
    cmp $0x0,%eax
    jne parent
    int $VANISH_INT
parent:
    mov %eax,%esi
    int $YIELD_INT       # yield(child_tid);
    int $VANISH_INT
```

A.4 double_thread_fork.S

```
#include <syscall_int.h>

.global main
main:
    int $THREAD_FORK_INT # fork 1st thread
    cmp $0x0,%eax
    je first_child
parent:
    int $VANISH_INT
first_child:
    int $THREAD_FORK_INT # fork 2nd thread
    int $VANISH_INT      # both threads vanish
```